高等学校计算机系列规划教材

汇编语言程序设计

（第 3 版）

丁　辉　主编

张丽虹　魏远旺　副主编

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书内容主要包括微机基础知识， Intel 8086/8088 指令系统，Intel 80x86、Pentium 增强和扩展指令，程序设计方法，高级汇编技术，系统功能调用，汇编语言与 C/C++的混合编程技术，上机操作方法。在程序设计各章中在给出一般例题的基础上，特别设置了综合举例章节；在系统功能调用、汇编语言与 C/C++的混合编程两章中更特地设置了实例章节。每章附有习题，书后附有上机实验指导。

本书可作为高等学校、高等职业学校计算机专业或相近专业汇编语言程序设计课程教材，微型计算机原理课程辅助教材，亦可供软件开发人员参考。

图书在版编目（CIP）数据

汇编语言程序设计/丁辉主编．—3 版．—北京：电子工业出版社，2009.3
高等学校计算机系列规划教材
ISBN 978-7-121-08033-3

Ⅰ. 汇… Ⅱ. 丁… Ⅲ. 汇编语言—程序设计—高等学校—教材 Ⅳ. TP313

中国版本图书馆 CIP 数据核字（2008）第 206398 号

策划编辑：吕 迈
责任编辑：吕 迈
印 刷：
装 订：
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1 092 1/16 印张：18 字数：461 千字
印 次：2009 年 3 月第 1 次印刷
印 数：4 000 册 定价：2 7 . 0 0 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

在众多程序设计语言中，汇编语言属于低级语言。“低级”主要是指在面向用户方面，汇编语言不及 C/C++、Java 等程序设计语言。而在面向机器方面，汇编语言之“高级”则无与伦比。汇编语言可以充分利用计算机的硬件特性，编制对时间和空间要求很高的程序，在实时控制场合，汇编语言更是无可替代，由此决定了汇编语言程序设计是计算机专业及相近专业人员的必备知识。

本书以 Intel 8086/8088 系列微机作为基础机型介绍汇编语言程序设计知识。在介绍 8086/8088 CPU 寻址方式和指令系统的基础上，详细介绍了汇编语言程序设计的基本方法和技巧，掌握这些内容，可以为 Intel 80x86 及 Pentium 系列微机的汇编语言程序设计奠定基础。考虑到 Intel 80x86 及 Pentium 系列微机的广泛应用，本书设置了关于 Intel 80x86 及 Pentium 的增强和扩展指令内容，在介绍各种程序设计方法的例题中也兼顾了这些指令的应用。本书的主体内容为 Intel 8086/8088 指令系统和各种程序设计方法，第 1 章和第 2 章则提供了学习汇编语言的基础知识，第 9 章和第 10 章提供了进行高效率、大规模汇编语言程序设计的必备知识；第 11 章讲述了用 C/C++进行混合编程的基本方法。

本书以编者长期使用的该课程讲稿为主体，以前两版本为基础，进行了系统的整合和内容的扩充，力求难点分散、循序渐进，为避免大量的汇编语言指令集中堆砌，将部分指令融于相关程序设计方法的介绍之中。对于同类内容讲透一点，以点带面。例题和习题的设置力图紧扣重点，举一反三，不仅有一般例题，更有综合举例和应用实例。每项实验均设有验证和设计两种类型的实验题，以便读者在巩固书本知识的基础上，提高应用和创新的能力。

本书由丁辉主编，张丽虹、魏远旺为副主编。第 5 章至第 9 章，以及上机实验指导由丁辉编写；第 1 章至第 4 章由张丽虹编写；第 10、11 章由魏远旺编写；全书由丁辉统稿。陈书谦、伍俊明、傅扬烈、姜宏岸、邵峥嵘、冯亚东、常赵罡为本书的编写提出了不少有益的建议，并参与了资料的整理工作。在编写过程中参考了相关书籍，包括书后参考文献中未能列出者，在此对相关作者表示诚挚的谢意。由于编者水平有限，书中难免存在疏漏，敬请同行专家指正。

在本书的编写过程中，得到了电子工业出版社的热情支持，在此一并表示衷心的感谢。

编者

2008 年 10 月

目　　录

第1章　基 础 知 识

本章提供了学习汇编语言程序设计所需的一些基础知识。首先对汇编语言程序设计进行了概述，其次对计算机常用的几种数制及其相互间的转换方法进行了讲解，并且介绍了数值数据和非数值数据在计算机中的表示方法。

1.1　汇编语言与汇编语言程序设计

1.1.1　汇编语言

计算机程序设计语言是人机交流的重要工具，可分为机器语言、汇编语言和高级语言。

机器语言是机器指令的集合，是一种面向机器的程序设计语言。机器指令是由 0 和 1 构成的二进制代码，不同种类的计算机具有各自的机器语言。其优点是可为计算机直接接受，用其编写的机器语言程序执行速度快，占内存空间小，可充分利用计算机的硬件特性；缺点是指令难记，用其编写的机器语言程序难以阅读且通用性差。

高级语言是面向问题求解过程或面向对象的程序设计语言。典型的高级语言有 Pascal，FORTRAN，C++，Java 等。高级语言接近于人类的自然语言，而且通用于各种计算机。其优点是易学易记，用其编写的高级语言程序易读易改，通用性强；其缺点是高级语言程序需经过编译或解释方能被计算机接受，执行速度慢，占内存空间大，不能直接利用计算机的硬件特性。

汇编语言又称为符号语言，实际上是一种符号化的机器语言。它将机器指令的操作码、操作数由二进制代码改为人们所熟悉的符号，例如

ADD　AL，5

表示将数字 5 加到 AL 中。汇编语言程序需经过汇编才能为计算机接受，这一点不如机器语言方便。虽然所用符号为人们所熟悉，然而不如高级语言那样接近人类的自然语言，程序编写和交流较为困难。除此以外，汇编语言几乎具备了机器语言的所有优点，一定程度上弥补了机器语言的缺陷，而且不存在高级语言的上述缺点。可以认为，汇编语言是目前使用的唯一直接利用计算机硬件特性的程序设计语言。

1.1.2　汇编语言程序设计

汇编语言程序设计是指使用汇编语言设计程序的过程。为什么要学习汇编语言程序设计？其原因至少有以下几点。

（1）通过汇编语言程序设计，人们可以高效地使用计算机解决现实问题。在解决同一现实问题时，汇编语言程序与高级语言程序相比，占用内存更小，执行速度更快。

（2）通过汇编语言程序设计，人们可以直接利用计算机的硬件特性，准确计算解决某一问题所需的时间，从而可实现实时控制。这一点是高级语言程序难以替代的。

（3）进行汇编语言程序设计，必然要从原理上认识、理解计算机的工作过程。因此，学

习汇编语言程序设计不仅可以掌握一种高效的程序设计方法，而且对于学习计算机组成原理、计算机原理也大有帮助。

1.2 进位计数制

进位计数制是计数的一种方法。人们普遍习惯的进位计数制为十进制。十进制数的基为10，有10个数码：0、1、2、3、4、5、6、7、8及9，遵循逢十进一的规则。二进制数的基为2，有2个数码：0、1，遵循逢二进一的规则。以此类推，N进制数的基为N，有N个数码：0、1、2、…、$N-1$，遵循逢N进一的规则。常用的几种进位计数制的情况如表1.1所示。

表1.1 常用的进位计数制

数　制	基	数　码	尾　标
十六进制	16	0，1，2，3，4，5，6，7，8，9，A，B，C，D，E，F	H
十进制	10	0，1，2，3，4，5，6，7，8，9	D（或默认）
八进制	8	0，1，2，3，4，5，6，7	Q（或O）
二进制	2	0，1	B

1.2.1 常用计数制及其数的算术运算

1. 十进制（Decimal）

首先观察以下十进制数的组成：

$$361.905D=3\times10^{2}+6\times10^{1}+1\times10^{0}+9\times10^{-1}+0\times10^{-2}+5\times10^{-3}$$

不难看出每一个数码根据它在这个数中所处的位置（数位）来决定实际数值。事实上，任意一个正的十进制数$S=K_{n-1}K_{n-2}\cdots K_0K_{-1}K_{-2}\cdots K_{-m+1}K_{-m}$都可以表示成以下形式：

$$S=K_{n-1}\times10^{n-1}+K_{n-2}\times10^{n-2}+\cdots+K_1\times10^{1}+K_0\times10^{0}+K_{-1}\times10^{-1}+\cdots+K_{-m+1}\times10^{-m+1}+K_{-m}\times10^{-m}$$

$$=\sum_{i=-m}^{n-1}k_i\times10^{i}$$

其中K_i可以是0至9这十个数码中的任意一个，m和n均为自然数，10^i称为权。一般而言，一个正的N进制数$S=K_{n-1}K_{n-2}\cdots K_0K_{-1}K_{-2}\cdots K_{-m+1}K_{-m}$可以表示为以下通用形式：

$$S=\sum_{i=-m}^{n-1}k_i\times N^{i}$$

其中K_i可以是0至$N-1$中任意一个数字，m和n均为自然数，N^i为各数位相应的权。

2. 二进制（Binary）

在日常生活中，存在着大量的两种对立的现象，例如：是和非，开和关，通和断。电子元件、物理器件中两种状态易于实现，如电压、电流的有和无，晶体管的导通和截止，这两种状态是非常稳定的，而找到8种、10种或16种稳定的状态则要复杂得多，所以计算机中采用二进制作为进位数制，以便于存储及计算的物理实现。

二进制数与人们常用的十进制数的对照关系如表1.2所示。

一个二进制数也可以用上述通用形式来表示。例如：

$$11101.011B=1\times2^4+1\times2^3+1\times2^2+0\times2^1+1\times2^0+0\times2^{-1}+1\times2^{-2}+1\times2^{-3}$$

二进制数的算术运算与十进制数的算术运算类似，区别仅在于该运算遵循逢二进一的规则。

【例 1.1】（1）1101B+1011B=11000B

$$\begin{array}{r} 1101 \\ +1011 \\ \hline 11000 \end{array}$$

（2）1101B−1011B=0010B

$$\begin{array}{r} 1101 \\ -1011 \\ \hline 0010 \end{array}$$

（3）1101B×1011B=10001111B

$$\begin{array}{r} 1101 \\ \times1011 \\ \hline 1101 \\ 1101 \\ 0000 \\ 1101 \\ \hline 10001111 \end{array}$$

（4）1111B÷101B=11B

$$\begin{array}{r} 11 \\ 101\overline{)\,1111} \\ 101 \\ \hline 101 \\ 101 \\ \hline 0 \end{array}$$

表 1.2　常用进位计数制对照表

十　进　制	二　进　制	八　进　制	十 六 进 制
0	000	0	0
1	001	1	1
2	010	2	2
3	011	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

3. 十六进制（Hexadecimal）

十六进制数采用 0～9，A～F 十六个数码，这里 A 表示 10D，B 表示 11D，C 表示 12D，D 表示 13D，E 表示 14D，F 表示 15D。

十六进制数与人们常用的十进制数的对照关系如表 1.2 所示。

一个十六进制数也可以用上述通用形式来表示。例如

$$6B.0CH=6\times16^1+B\times16^0+0\times16^{-1}+C\times16^{-2}$$

十六进制数的算术运算与十进制数的算术运算类似，区别仅在于该运算遵循逢十六进一的规则。

【例 1.2】（1）8A04H+110CH=9B10H　　（2）8A04H−110CH=78F8H

8A04	8A04
+ 110C	− 110C
9B10	78F8

在汇编语言程序中，数据通常采用十六进制形式，汇编语言的调试、列表文件中显示的数据也都是使用十六进制形式，所以有必要熟练掌握这种数制及其数的运算。

4. 八进制（Octal）

八进制数采用 0～7 八个数码，与十进制数的前八个数码一一对应，如表 1.2 所示。

一个八进制数也可以用上述通用形式来表示。例如

$$36.53Q=3\times8^{1}+6\times8^{0}+5\times8^{-1}+3\times8^{-2}$$

1.2.2 数制转换

1. 非十进制数→十进制数

非十进制数转换为十进制数的方法是：将非十进制数用上述通用形式表示，即按权展开，计算的结果即为十进制数。不管非十进制数是否带有小数部分，均可用这种方法转换。

【例 1.3】$1010110B=1\times2^{6}+0\times2^{5}+1\times2^{4}+0\times2^{3}+1\times2^{2}+1\times2^{1}+0\times2^{0}=86D$

$4D.8H=4\times16^{1}+13\times16^{0}+8\times16^{-1}=77.5D$

$1362Q=1\times8^{3}+3\times8^{2}+6\times8^{1}+2\times8^{0}=754D$

2. 十进制数→非十进制数

十进制数转换为二进制、八进制及十六进制等非十进制数的工作可分为整数的转换和小数的转换两种情况。一个十进制数与其对应的非十进制数相比，两者的整数部分和小数部分分别相等。于是，将一个十进制数转换为非十进制数时，可以对其整数部分和小数部分分别作转换，将两个转换结果结合起来就可以得到对应的非十进制数。

（1）十进制整数→非十进制整数

十进制整数转换为二进制、八进制及十六进制等非十进制整数可以采用除基取余法或减权记位法。前一方法较易掌握，但使用时较烦琐；后一方法使用时较方便，但需要熟悉非十进制各数位对应的权值。

① 除基取余法。将十进制整数及此间产生的商不断除以非十进制数的基，直至商为 0 为止，并记下每一次相除所得到的余数，按照从后往前的次序，将各余数作为 $K_{n-1}\,K_{n-2}\cdots K_0$，从而构成对应的非十进制数。

【例 1.4】将 233D 转换为十六进制数。

按照题目要求，基应取 16，具体转换过程如下：

	除以基 2	余数	K_i
16	233	9	K_0
16	14	14（即 E）	K_1
	0		

转换结果为：233D=E9H。

【例 1.5】将 233D 转换为二进制数。

按照题目要求，基应取 2，具体转换过程如下：

转换结果为：233D=11101001B。

	除以基 2	余数	K_I
2	233	1	K_0
2	116	0	K_1
2	58	0	K_2
2	29	1	K_3
2	14	0	K_4
2	7	1	K_5
2	3	1	K_6
2	1	1	K_7
	0		

② 减权记位法。减权记位法常用于十进制数到二进制数的转换，对于十进制数到其他进制数的转换使用这种方法则不够方便。这里仅介绍十进制数到二进制数转换时，减权记位法的具体内容。

将十进制整数与其最相近的权值 2^{n-1} 做比较，前者不小于后者则减去 2^{n-1}，并在 $n-1$ 位记 1；否则在 $n-1$ 位记 0。然后再与 2^{n-2} 做比较并做相同的工作，直至最低位被记为 1 或 0。从 $n-1$ 位、$n-2$ 位直至最低位所记的 1 或 0 就构成了二进制数 $K_{n-1}K_{n-2}\cdots K_0$。通常第 1 次比较所用的 2^{n-1} 取做小于等于十进制整数的最大二进位权值。

【例 1.6】将 233D 转换为二进制数。

由于 $2^7<233D<2^8$，故开始用做比较的数值取 2^7。

具体转换过程如下：

比较，减权	记位	K_i
$233-2^7=105$	1	K_7
$105-2^6=41$	1	K_6
$41-2^5=9$	1	K_5
$9<2^4$	0	K_4
$9-2^3=1$	1	K_3
$1<2^2$	0	K_2
$1<2^1$	0	K_1
$1-2^0=0$	1	K_0

转换结果为：233D=11101001B。

【例 1.7】将 130D 转换为二进制数。

考虑到 130D 为 128D 与 2D 之和，即为 2^7 与 2^1 之和，于是对应的二进制数 $K_7K_6\cdots K_0$ 中 K_7=1，K_1=1，其余位均为 0，也即转换结果为 130D=10000010B。

由该例可见，在熟悉二进制各数位对应的权值的基础上，参照减权记位法可以方便地进行十进制整数到二进制整数的转换。

（2）十进制小数→非十进制小数

十进制小数转换为二进制、八进制及十六进制等非十进制小数可以采用乘基取整法或减权记位法。前一方法较易掌握，但使用时较烦琐；后一方法使用时较方便，但需要熟悉非十进制数各数位对应的权值。

① 乘基取整法。将十进制小数以及此间产生的小数部分不断乘以非十进制数的基，并记下每次相乘所得到的整数部分，直至积的小数部分为 0 为止，按照从前往后的次序，将各整数部分作为 K_{-1}，K_{-2}，… K_{-m}，就构成了对应的非十进制数 $K_{-1}K_{-2}\cdots K_{-m}$。

【例 1.8】将 0.6875D 转换为二进制数。

按照题目要求，基应取 2，具体转换过程如下：

$$
\begin{array}{rl}
 & 0.6875 \\
 & \times\quad 2 \\
\hline
K_{-1} \leftarrow & \boxed{1}.3750 \\
 & \times\quad 2 \\
\hline
K_{-2} \leftarrow & \boxed{0}.7500 \\
 & \times\quad 2 \\
\hline
K_{-3} \leftarrow & \boxed{1}.5000 \\
 & \times\quad 2 \\
\hline
K_{-4} \leftarrow & \boxed{1}.0000
\end{array}
$$

转换结果为：0.6875D=0.1011B。

【例 1.9】将 0.6875D 转换为八进制数。

按照题目要求，基应取 8，具体转换过程如下：

$$
\begin{array}{rl}
 & 0.6875 \\
 & \times\quad 8 \\
\hline
K_{-1} \leftarrow & \boxed{5}.5000 \\
 & \times\quad 8 \\
\hline
K_{-2} \leftarrow & \boxed{4}.0000
\end{array}
$$

转换结果为：0.6875D=0.54Q。

当一个十进制小数对应的非十进制小数的位数过多时，可根据需要取前若干位作为近似结果。

② 减权计位法。与上述十进制整数转换为二进制整数时使用的减权计位法类似，但有两点区别：其一，十进制小数首先应与 2^{-1} 比较，然后与 2^{-2} 比较，依次类推；其二，有时要根据需要，取部分小数位作为近似转换结果。

【例 1.10】将 0.6875D 转换为二进制数。

具体转换过程如下：

比较、减权	记　位	K_I
$0.6875-2^{-1}=0.1875$	1	K_{-1}
$0.1875<2^{-2}$	0	K_{-2}
$0.1875-2^{-3}=0.0625$	1	K_{-3}
$0.0625-2^{-4}=0$	1	K_{-4}

转换结果为：0.6875D=0.1011B。

如前所述，将一个十进制数转换为非十进制数时，可以对其整数部分和小数部分分别作转换，然后将整数部分和小数部分的转换结果结合成为最终结果。

【例 1.11】将 233.6875D 转换为二进制数。

由前面的例题可知：

233D=1110 1001B

0.6875D=0.1011B

于是转换结果为：233.6875D=11101001.1011B。

3. 二进制数↔八进制数、十六进制数

由于一位八进制数对应三位二进制数，一位十六进制数对应四位二进制数，于是二进制数与八进制数、十六进制数之间的转换比较简单。

（1）二进制数→八进制数、十六进制数

将二进制数由小数点向左右每三位分为一组（不足三位则用 0 补充），每一组用对应的八进制数码表示，即可得到对应的八进制数。类似地，将二进制数由小数点向左右每四位分为一组（不足四位则用 0 补充），每一组用对应的十六进制数码表示，即可得到对应的十六进制数。

【例 1.12】将 0101 1101.01B 分别转换为八进制数和十六进制数。

01 011 101.01B = 001 011 101 .010B = 135.2Q

0101 1101.01B = 0101 1101.0100B = 5D.4H

说明：在分组中若不足三位或不足四位时，一定要用 0 补充，否则极易出错。就该例而言，若不注意这一点，就极易将结果错写为 135.1Q 和 5D.1H。

（2）八进制数、十六进制数→二进制数

将八进制数的每一位数用三位二进制数码表示，即可得到对应的二进制数。类似地，将十六进制数的每一位数用四位二进制数码表示，即可得到对应的二进制数。必要时可以去掉转换结果中的前 0 和尾 0。

【例 1.13】分别将 45.4Q 和 27CH 转换为二进制数。

45.4Q=100 101 .100B=100101.1B

27CH=0010 0111 1100B=100111 1100B

从以上介绍可以看出，在将十进制数转换为非十进制数时，转换为二进制数较为简单，转换为八进制数、十六进制则较为复杂。考虑到二进制数转换为八进制数、十六进制数比较简单，故在需要将十进制数转换为八进制数、十六进制数时，可以先将其转换为二进制数，然后再由二进制数转换为八进制数、十六进制数。

【例 1.14】将 233.6875D 转换为十六进制数。

首先将 233.6875D 转换为二进制数，由【例 1.11】可知：

233.6875D = 1110 1001.1011B

然后将 11101001.1011B 转换为十六进制数。

1110 1001.1011B = E9.BH

于是十进制数到十六进制数的转换结果为：233.6875D = E9.BH

1.3 计算机中数和字符的表示

人们在使用计算机时，需要计算机不仅能处理数，即数值数据，而且能处理非数值数据，如字符、声音和图像等。虽然非数值数据在计算机中也是以二进制代码的形式存储，但它们的实际意义与数值数据不同。

1.3.1 数的表示

1. 机器数与真值

在本书前面的介绍中，均将数视为无符号数。例如，1010 0010B 的每一位二进制数码均被视为数值，其对应的十进制数为 162。有符号数在计算机中如何表示？首先需要解决数符如何表示的问题。在计算机中实际采用数值化的方法来表示数符，通常用 0 表示正，用 1 表示负。这样表示的数称为机器数，而人们所习惯的用+、−分别表示正、负的数称为真值。机器数的位数与计算机的字长有关，例如

真值	机器数（假设使用 8 位表示）	
+10110	0	0010110
−11010	1	0011010

这里机器数采用原码表示。实际上，机器数还可以采用补码、反码表示。

2. 原码

原码的最高位表示真值的数符，其余位为数值位，且与真值的数值位相同，必要时在数值位前加上前 0。

数的原码表示具有直观、与真值的转换方法简单等优点，但是原码有着算术运算复杂的缺陷。与手算相同，做加法运算时，首先要判断两数的数符，同号则相加，异号则相减。做减法运算时，还必须比较两数的绝对值，用较大的绝对值减去较小的绝对值，差的数符则采用绝对值较大者的数符。以上的算法对于硬件实现来说比较困难。

3. 补码

补码的引入，是为了简化减法运算。补码的概念在日常生活中经常用到，例如手表的校时。假定手表停在 11 时，而标准时间为 9 时，可以使用两种校时方法：一种方法是逆时针调 2 小时，即 11−2=9；另一种方法是顺时针调 10 小时，在调至 12 时后则为 0 时，即到 12 时时归零，因此有 11+10=9（称为模加）。于是，就手表校时而言有：

$$11-2 = 11+10 \qquad (\text{MOD } 12)$$

这里 12 称为模，10 称为 2 的补数。由此看出，减去一个数等价于加上该数的补数。下面介绍计算机中使用的补码的含义、求取方法及其运算。

（1）补码定义

$$[X]_{补} = \begin{cases} X & 0 \leqslant X < 2^{n-1} \\ 2^n - |X| & -2^{n-1} \leqslant X < 0 \end{cases} \qquad (\text{MOD } 2^n)$$

当一个数为正数，则其补码就是该数本身；为负数，则其补码等于模值与该数绝对值之差。式中的 n 为补码的位数。

（2）根据真值求补码

根据补码定义可以求取一个数的补码，然而还可以采用更为简便的方法。求取一个数的 n 位补码的简便方法是：对于正数，通过补前 0，将其数值部分补至 n 位即可；对于负数通过补前 0，将其数值部分补至 n 位，然后按位取反并在末位加 1 即可。

【例 1.15】求出以下各数的 8 位补码：

+1000011B，−111000B，+1111111B，−10000000B，0

① 将+1000011B 的数值部分通过补前 0 达到 8 位，即可得到其补码：

$[+1000011B]_{补}$=01000011B

② 将−111000B 的数值部分通过补前 0 达到 8 位，即 00111000B。

按位取反后得到：11000111B

末位加 1 后得到：11001000B

$[-111000B]_{补}$=11001000B

③ 将+1111111B 的数值部分通过补前 0 达到 8 位，即可得到其补码：

$[1111111B]_{补}$=01111111B

④ −10000000B 的数值部分已达 8 位，即 10000000B。

按位取反后得到：01111111B

末位加 1 后得到：10000000B

$[-10000000]_{补}$=10000000B

⑤ 用同样方法不难求得：

$[0]_{补}$=0

（3）根据补码求真值

补码最高位为 0，则真值数符为“+”，真值数值位与补码其余位相同；补码最高位为 1，则真值数符为“−”，将补码所有位按位取反且末位加 1 后就可得到真值的数值位。

用这种方法可以方便的将【例 1.15】中求得的补码还原为对应的真值。

（4）补码的表示范围及补码的扩展

① 补码的表示范围。若要求给出+10000000B 和−10000100B 的 8 位补码，结果如何？显然，将求不出正确的结果。原因在于，这两个数超出了 8 位补码所能表示的范围−128～+127。一般而言，n 位补码所能表示的范围为-2^{n-1}～$+2^{n-1}-1$。在计算机中，n 常取 8，16，32 等。以上两个数可以表示为 16 位补码：

$[+10000000B]_{补}$=0000 0000 1000 0000B

$[-10000100B]_{补}$=1111 1111 0111 1100B

② 补码的扩展。为了满足进行算术运算等方面的需要，有时要求将一个补码扩展成双倍位数，比如由 8 位补码扩展为 16 位补码，由 16 位补码扩展为 32 位补码等。扩展方法是：将扩展的各位都置为原来补码的最高位。

例如：$[+100\ 0011B]_{补}$=0100 0011B=0000 0000 0100 0011B

$[-11\ 1000B]_{补}$=1100 1000B=1111 1111 1100 1000B

（5）补码的加减法运算

补码加法规则：$[X+Y]_{补}=[X]_{补}+[Y]_{补}$

补码减法规则：$[X-Y]_{补}=[X]_{补}+[-Y]_{补}$

由此可见，使用补码可以将减法运算转换成加法运算，避免了前述减法运算中的困难。

【例 1.16】补码加法运算。

	十进制运算	补码运算
①	41	00101001
	+27	+00011011
	68	01000100
②	41	00101001
	+（−27）	+11100101
	14	100001110
③	−41	11010111
	+27	+00011011
	−14	11110010
④	−41	11010111
	+（−27）	+11100101
	−68	110111100

上述②、④中从最高位向前的进位由于位数的限制而自动丢失，但这并不影响运算结果的正确性。

【例 1.17】补码的减法运算。

	十进制运算	补码运算
①	41	00101001
	−27	+11100101
	14	100001110
②	41	00101001
	−（−27）	+00011011
	68	01000100
③	−41	11010111
	−（−27）	+00011011
	−14	11110010
④	−41	11010111
	−27	+11100101
	−68	110111100

在计算机中，补码减法是通过对减数求补后把减法转换为加法进行的。①、④中的最高位向前位的进位同样自动丢失，而不影响运算的结果。

4. *反码*

反码的最高位表示真值的数符，0 表示正，1 表示负。当反码最高位为 0，则其余位与真值的数值位相同；当反码最高位为 1，则其余位是真值的数值位按位取反后的结果。

5. BCD码

十进制小数和二进制小数相互转换时可能产生误差，而且日常习惯使用的是十进制，计算机中内部采用的是二进制。为方便十进制和二进制之间的转换，计算机允许内部采用一组四位二进制数来表示一位十进制数，组间仍然采用“逢十进一”的规则进行。这种采用二进制表示的十进制数编码称为 BCD 码（Binary Coded Decimal）码。该编码的方法和使用详见本书第五章。

1.3.2 字符的表示

计算机只能识别二进制数，因此计算机中的数字、字母、符号、控制符、汉字等也必须采用二进制进行编码。

1. ASCII码

非汉字常用字符包括：

字母：A，B，…，Z；a，b，…，z

数字：0，1，…，9

专用符号：+，−，*，/ …

为了使计算机硬件能够识别和处理这些字符，必须对字符按一定规则用二进制编码，目前广泛使用的是 ASCⅡ码（美国国家标准信息交换码见附录 B），这种代码用一个字节表示一个字符，最高位一般为校验位。

2. 汉字编码

有了 ASCII 码，计算机就能处理数字和英文字母等字符，但是还不能处理汉字字符。为了使计算机能够处理汉字信息，就必须对汉字也进行编码。

（1）GB 2312 汉字编码

我国于 1981 年颁布了第一个国家标准——《信息交换用汉字编码字符集・基本集》（GB 2312）。该标准选出 6763 个常用汉字和 682 个非汉字字符，为每个字符规定了标准代码，以便在不同的计算机系统之间进行汉字文本的交换。

（2）GBK 汉字内码扩充规范

GB 2312 只有 6763 个简体汉字，在许多处理中有很大缺憾，如户籍中的人名、地名等。GBK 是我国 1995 年发布的又一个汉字编码标准，一共有 21003 个汉字和 883 个图形符号，与 GB 2312 国标汉字字符集及其内码保持兼容，增加了大量了繁体字和符号。

（3）GB 18030 汉字编码标准

2000 年我国政府发布新的汉字国家标准，解决 GBK 与台湾和香港地区 Big5 汉字编码不兼容的问题，同时也与国际标准化组织（ISO）制订的全球集中统一编码 UCS-2 接轨，保护了已有的大量中文资源。GB 18083 标准与 GB 2312 标准和 GBK 标准保持向下兼容，扩充了 UCS 中其他字符。

习　题

1.1　将下列十进制数转换为二进制数和十六进制数。

（1）（369）D =（　）B =（　）H

（2）（355）D =（　）B =（　）H

（3）（127）D =（　）B =（　）H

（4）（1000）D =（　）B =（　）H

1.2　将下列二进制数转换为十六进制数和十进制数。

（1）（1011110111）B=（　）H=（　）D

（2）（11000100101）B=（　）H=（　）D

（3）（10000000）B=（　）H=（　）D

（4）（11100101）B=（　　）H=（　）D

1.3　下表给出了十进制数，请写出与之对应的二进制真值和 8 位补码。

表示方式 / 十进制数	真　值	补　码
−31		
+72		
−1		
−128		

1.4　下列各数均为十进制数，请用 8 位补码计算下列各题，并用十六进制数表示其运算结果。

1）69−48　　2）−69+48　　3）−69−48

1.5　分别将下列各数看作无符号数和补码，则各自对应的十进制数是什么？

1）98H　　2）31H　　3）FFH　　4）80H

1.6　如果用 16 位存储一个无符号数，这个数的范围是什么?如果存储放入是一个补码表示的有符号数，那么这个数的范围是什么?

1.7　请写出下列字符串的 ASCII 码值。

'Hello'

'Please　give　me　$10. '

第 2 章　IBM-PC计算机系统概述

汇编语言是一种符号化的面向机器的语言，所以介绍汇编语言编程需要基于一种机器模型。本章主要以 Intel 处理器为基础从 CPU 的内部结构、寄存器的功能、存储器组织以及外设等方面讲述 IBM-PC 计算机系统的构成。

2.1　Intel 8086/8088 CPU的功能结构

CPU 在处理存储在存储器中的程序时，总是有规律地按以下步骤工作：

（1）从存储器中取出一条指令；

（2）分析指令的操作码；

（3）（如果需要）从存储器读取操作数；

（4）执行该指令；

（5）（如果指令要求）将结果写入存储器；

（6）重复（1）～（5）。

8086/8088 CPU 采用了流水线结构，它将上述步骤分配给 CPU 内两个独立的处理单元：执行单元（EU）和总线接口单元（BIU）。执行单元主要负责指令的分析与执行工作；接口部件单元主要负责取指令和存取操作数。这两个单元并行工作，在分析和执行指令的同时进行存取操作。由于 EU 执行的是 BIU 事先从存储器中取出的指令，多数情况下取指令的时间“消失”了，从而加快了程序的运行速度。8086/8088 CPU 的功能结构如图 2.1 所示。

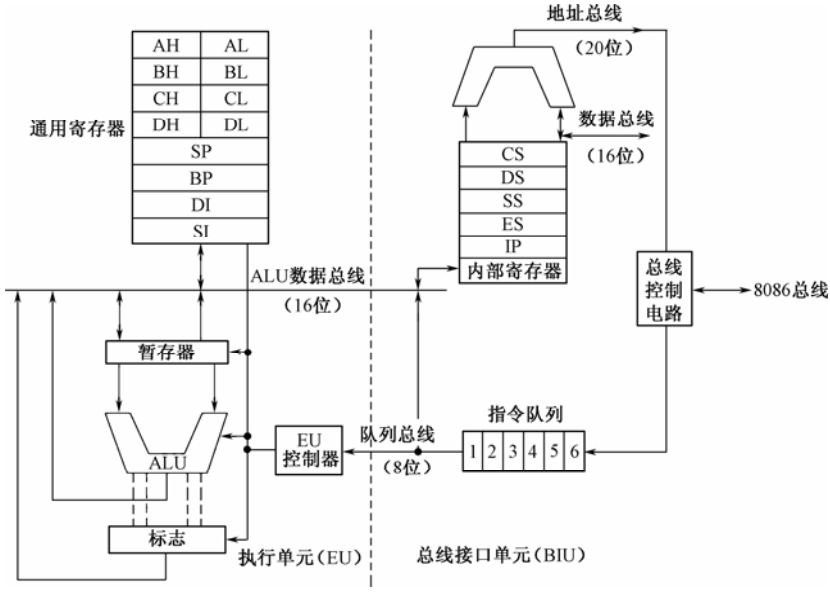


图 2.1　8086/8088 CPU 结构框图

2.1.1 执行单元与接口部件单元

执行单元 EU 包括一个 16 位算术/逻辑单元 ALU，一个 16 位状态标志寄存器，一组数据寄存器，一组指针和变址寄存器以及数据暂存器和 EU 控制电路。所有的寄存器和数据线都是 16 位的，以加快内部的传送过程。EU 不直接同外部总线相连，它从总线接口单元 BIU 的指令队列中获取指令。执行指令时，若需要访问存储器或外部设备，EU 就向 BIU 发出操作请求，由 BIU 完成相应的操作。

总线接口单元 BIU 包括一组 16 位段寄存器（CS、DS、ES、SS），一个 16 位的指令指示器，指令队列，地址加法器和总线控制电路。其中指令队列是一个内部的 RAM 阵列，它类似一个先进先出的栈。EU 执行指令过程中，BIU 始终“提前”从存储器中预取出下面可能被执行的指令填入队列，若指令队列中出现空字节，且 EU 又没有请求 BIU 进行数据的存取操作，BIU 就会自动执行取指令操作，以填满指令队列。大多数情况，指令队列中始终有指令，所以 EU 不必等待 BIU 取指令，若 EU 执行的是一条转移指令，则 BIU 会使指令队列复位，从新的地址取出指令并直接送给 EU 执行，接着 BIU 会取入后续指令来重填指令队列。

寄存器在计算机中起着非常重要的作用。每个寄存器相当于运算器中的一个存储单元，但它的存取速度比存储器快得多，可以与 ALU 等部件保持同步。寄存器主要用于存储计算过程中所需的信息或计算的中间、最终结果。

8086/8088CPU 内部工作寄存器包括 14 个 16 位的寄存器。它们按功能可分为通用寄存器、控制寄存器组和段寄存器组。寄存器结构如图 2.2 所示。

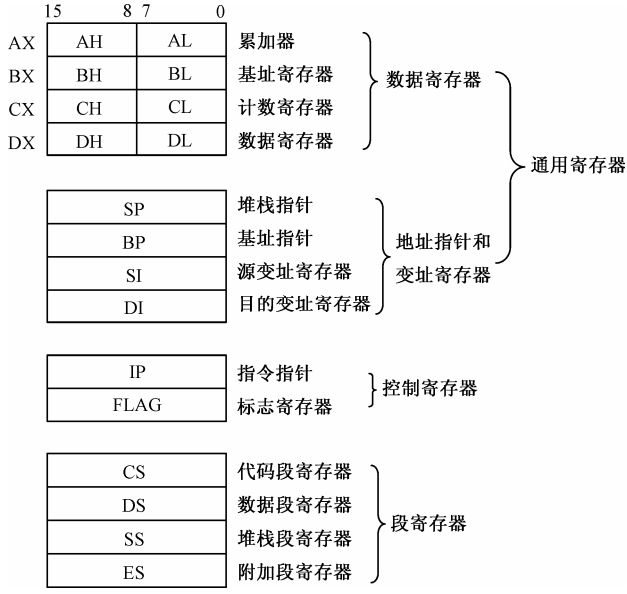


图 2.2　寄存器结构图

1. 通用寄存器

通用寄存器包括数据寄存器、地址指针和变址寄存器。

（1）数据寄存器

数据寄存器 AX、BX、CX、DX 主要用来保存算术运算或逻辑运算的操作数、中间结果。

这些寄存器的存在，避免了每一次运算都需要访问存储器，从而节约 CPU 的时间。以上这四个寄存器既可以作为一个 16 位寄存器使用，也可以分别作为两个 8 位的寄存器使用。AX 可分为 AH（高字节 8 位）和 AL（低字节 8 位），BX 可分为 BH、BL，CX 可分为 CH、CL，DX 可分为 DH、DL。数据寄存器的双重性使得 8086/8088 非常容易处理字和字节数据。

数据寄存器既有通用性，又有专用性，其用途如表 2.1 所示。

表 2.1　数据寄存器的用途

寄　存　器	用　　途
AX	作为累加器，是算术运算的主要寄存器，另外所有 I/O 指令均使用该寄存器与外部设备交换信息
BX	可以用于提供基址
CX	在循环（LOOP）和串处理指令中用做隐含的计数器
DX	一般双字长运算时把 DX 和 AX 组合在一起存放一个双字长数，DX 用来存放高位字。此外，对于某些 I/O 操作，DX 可以用来存放 I/O 的端口地址

指令中规定的专门用途，可以缩短指令代码长度或者使这些寄存器具有隐含的性质，以简化指令的书写形式。根据数据寄存器的某些专门用途，又常把 AX、BX、CX、DX 分别称为累加器，基址寄存器，计数寄存器和数据寄存器。

（2）地址指针及变址寄存器

包括基址指针 BP、堆栈指针 SP 以及变址寄存器 SI 和 DI。它们主要用于保存段内的偏移量，也可以像数据寄存器一样在运算过程中存放操作数，但只能以字（16 位）为单位使用。做段内寻址时，SP 称为堆栈指针寄存器，BP 称为基址指针寄存器，它们都可以与 SS 段寄存器一起确定堆栈中的某一存储单元的物理地址。SP 用来指示栈顶的偏移地址，堆栈操作的 PUSH 和 POP 指令均从 SP 寄存器得到现行堆栈段内的偏移量。BP 可作为堆栈区中的一个基地址，以便访问堆栈中的某一存储单元。SI（源变址寄存器）、DI（目的变址寄存器）常用于字符串操作指令中保存操作数的偏移量。源操作数的偏移量存放于 SI 中，而目的操作数的偏移量存放于 DI 中。其用途如表 2.2 所示。

表 2.2　指针和变址寄存器的用途

寄　存　器	用　　途
BP	可用于提供基址
SP	在堆栈操作中作为堆栈指针
SI	在字符串指令中作源变址寄存器；在间接寻址中作为地址寄存器；在间接寻址中作为变址寄存器
DI	在字符串指令中作目的变址寄存器；在间接寻址中作为地址寄存器；在间接寻址中作为变址寄存器

2. 段寄存器

段寄存器用来存放段地址。段寄存器组包括代码段寄存器 CS，数据段寄存器 DS，堆栈段寄存器 SS 和附加段寄存器 ES。

在 8086/8088 系统中，把 1MB 的存储空间划分为若干个逻辑段，段地址存放在段寄存器中，其中 CS 存放当前代码段的段地址，指令总是从这个段取出；SS 存放当前堆栈的段地址，堆栈操作所处理的就是此段中的数据；DS 存放当前数据段的段地址，指令中所涉及的存储器操作数就存放在此段中；ES 存放当前附加段的段地址，附加段通常也用来存放存储器的操作数。关于段的概念将在第 2.2 节进一步讨论。

3. 指令指示器IP

指令指示器 IP 是一个 16 位的寄存器，它用于存放代码段中指令的偏移地址，IP 总是包含下一条要取出的指令或当前指令的下一个字节机器码的偏移地址。根据 CS 中存放的代码段地址和 IP 的内容，形成待取指令在存储器中的 20 位物理地址。程序在顺序执行期间，IP 存放由 BIU 取出的指令代码的偏移量。每当取出一个指令的一字节机器码后，IP 会自动加 1。应当注意，程序员不能直接访问 IP 的内容，但可以通过程序指令，来改变程序的执行顺序，其实质就是间接改变 IP 的值。

4. 程序状态字寄存器PSW

PSW（program status word）是一个 16 位的寄存器。8086/8088 CPU 只使用了其中的 9 位，这 9 位可以分为两类：状态标志和控制标志，状态标志包括 CF、PF、AF、ZF、SF 和 OF，其余的 TF、IF、和 DF 为控制标志（如图 2.3 所示）。状态标志反映了 EU 执行算术或逻辑运算后的结果特征，这些特征常用来影响或控制某些后续指令执行；控制标志用来控制 CPU 的某些操作，它们可用程序设置或清除。

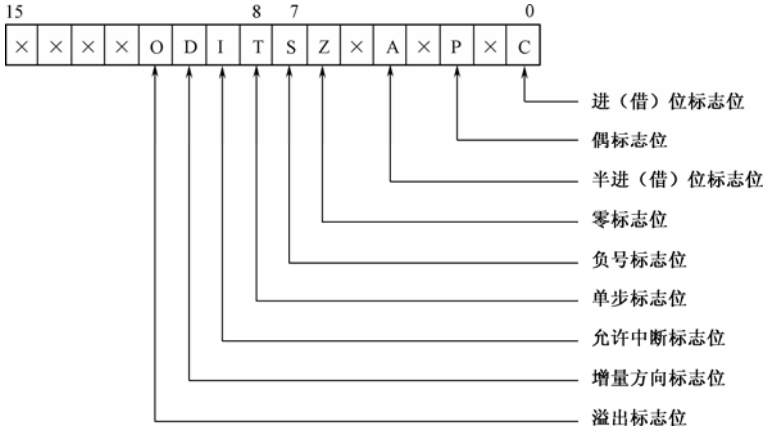


图 2.3　程序状态字寄存器

（1）状态标志

CF（Carry Flag），进（借）位标志，记录算术运算时从数字有效位产生的进位或借位值。例如：执行加法或减法指令时，结果的数字有效位有进位或有借位，则 CF=1；否则 CF=0。另外，循环移位指令执行过程也会改变此标志。

PF（Panty Flag），偶标志，此标志反映了运算结果中“1”的个数是奇数或是偶数。当运算结果中“1”的个数为偶数时，PF=1；否则 PF=0。它能够为机器中传送信息时可能产生的代码出错情况提供检验条件。

AF（Auxiliary Flag），半进（借）位标志，记录运算时第 3 位（半个字节）产生的进位值。例如：在执行加法运算过程中，若第 3 位有进位，或执行减法运算过程，第 3 位有借位，则 AF=1；否则 AF=0。该标志用于十进制算术运算指令。

ZF（Zero Flag），零标志，若运算结果为零，则 ZF=1；否则 ZF=0。

SF（Sign Flag），负号标志，它与运算结果的最高字有效位相同，并用来表示有符号数运算结果的符号。SF=1 表示运算结果为负；SF=0，表示运算结果为正。

OF（Overflow Flag），溢出标志，当有符号数运算结果产生了溢出，则 OF=1；否则 OF=0。

（2）控制标志

DF（Direction Flag），方向标志，在串处理指令中，用于控制处理信息的方向。当 DF=1 时，每次操作后使变址寄存器 SI 和 DI 减量，使串处理从高地址向低地址方向处理；当 DF=0 时，则使变址寄存器 SI 和 DI 增量，使串处理从低地址向高字地址方向处理。

IF（Interrupt Flag），中断标志，IF=1 时，允许 CPU 响应可屏蔽的外部中断请求；IF=0 时，CPU 对可屏蔽外部中断请求不予响应。

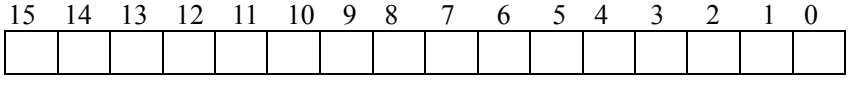
TF（Trap Flag），陷阱标志，用于单步方式操作。若 TF=1，使 CPU 在每一条指令执行结束后，会自动产生一次内部中断，CPU 转去执行一个中断服务程序。这种方式用于程序的测试过程。

以上是 PSW 中各状态位和控制位的含义，其中状态标志是由 CPU 根据计算的结果自动设置的，控制标志则是由系统程序或用户程序根据需要用指令来设置。

2.2 Intel 8086/8088 存储器的组织

2.2.1 存储单元的地址和内容

计算机存储器的基本单位是一个二进制位，每位可存储一个二进制数“0”或“1”，每 8 位组成一个字节，使用 8086/8088 CPU 的计算机系统字长为 16 位，由 2 个字节组成，位编号如下：



为了能正确存储或取得信息，给每一个字节单元指定一个编号（这个编号是唯一的），CPU 就按照编号进行访问，该编号称为存储器地址，地址从 0 开始编号，顺序每次加 1。在机器里，地址是用无符号二进制数表示的，为书写方便，常用十六进制数表示。n 位二进制数可以表示 2^n 个地址单元。为了方便，描述存储器容量大小的单位通常有：KB，1KB=2^{10} B=1 024B；MB，1MB= 2^{20} B =1 024×1 024B 等。65 536 个字节单元的存储容量为 64KB。

2.2.2 Intel8086/8088 存储器的组织

1. 存储器结构

8086/8088 CPU 具有 1MB（1 兆字节）的寻址能力，也即可以有 2^{20} 个存储单元来存放信息，但是由于外部数据总线宽度不同，存储器结构也不同。

8086 具有 16 位宽的数据总线，在访问存储器时，可以访问字，也可以访问字节。8086 的有些指令仅用于访问（读或写）一个字节数据，该字节数据可能存放于偶地址单元（如图 2.4 中 0002H 单元），也可能存放于奇地址单元（如图 2.4 中 0003H 单元）；而另一些指令用来访问字，即 16 位数据，16 位数据总是存放在相邻的两个单元内，且低位字节（低 8 位）总是存放在地址较低的单元，并把该单元地址称为该字的存储地址。例如：0002H 单元中存放的字数据为 6A47H；同样，0005H 单元中存放的字数据为 2B91H，如图 2.4 所示。

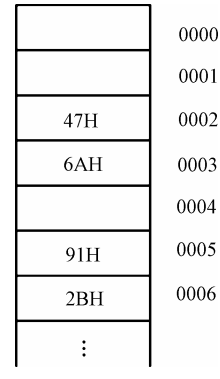


图 2.4 字数据存储格式

从上面可以看出，字地址可以是偶数（如 0002H），也可以是奇数（如 0005H）。通常把存放在偶地址的字称为规则字，存放在奇地址的字称为非规则字。8086CPU 的 BIU 是这样设计的，若访问一个字节的数据，用一个总线周期来完成操作；若访问一个字，则需用一个或两个总线周期来完成。对规则字，只需一个总线周期就可完成 16 位数据的传送，而对于非规则字则用相邻两个总线周期来完成该字的存取操作，这样会花费较多的时间。

2. 存储器的分段结构

对于 16 位字长的计算机，寄存器的长度为 8 或 16 位，16 位地址可以访问的最大存储空间为 64KB。而在使用 8086/8088 CPU 的计算机系统中，最大存储容量为 1MB。由于要访问 1MB 需要 20 位地址（2^{20}=1M），用十六进制数表示 1MB 的地址范围应为 00000H～FFFFFH，那么，在 16 位字长的机器里，如何能提供 20 位地址呢？在 8086/8088 CPU 的计算机系统中，采用了存储器地址分段的方法。

8086/8088 CPU 把 1MB 的存储器空间划分为任意的一些存储段，每个段的大小可达 64KB，这样段内的地址可以用 16 位二进制数来表示，也即可以在 CPU 的寄存器中进行处理。各段的大小可根据用途而定，可以是几个或几百个，乃至 64KB 范围内的任意个字节。段内地址连续，段与段是独立的。对段的起始地址有一定的限制，段不能从任意地址位置开始，必须是能够被 16 整除的地址，其特征是十六进制数表示的地址中，最低位为 0（即 20 位地址的低 4 位为 0）。

有了段地址是否就能直接访问存储器呢？在 1MB 的存储器中，每个存储器单元都有唯一的 20 位地址，称之为该存储单元的物理地址。CPU 在访问存储器时，必须先确定所要访问的存储器单元的物理地址才能取得该单元的内容。20 位物理地址由 16 位段地址和 16 位偏移地址组成。16 位段地址和 16 位偏移地址统称为逻辑地址，程序中所使用的地址是逻辑地址。CPU 在访问存储器时，根据段地址和偏移地址在 BIU 的地址加法电路中形成 20 位物理地址，其形成方法是将段地址左移 4 位后加偏移量，如图 2.5 所示。也即

段地址×16d+偏移量=物理地址

对应于存储器的每个单元均有一个唯一确定的物理地址，但逻辑地址不唯一，也即一个物理地址可对应多个逻辑地址。如图 2.6 所示，表示物理地址为 30403H 的存储单元，当段地址为 3000H 时，偏移地址为 0403H；当段地址为 3040H 时，偏移地址为 0003H。

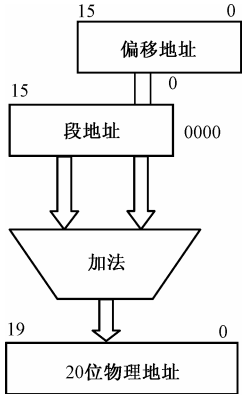


图 2.5　20 位物理地址的形成

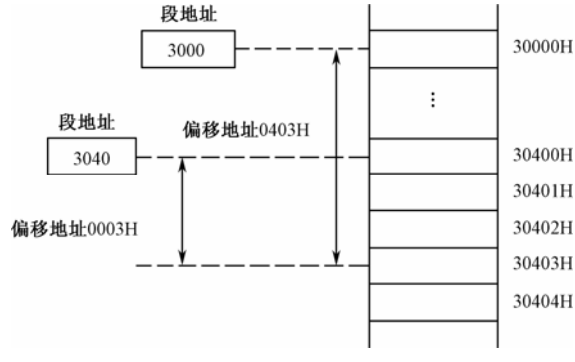


图 2.6　逻辑地址与物理地址

在 8086/8088 CPU 中，有 4 个段寄存器：CS、DS、SS 和 ES。这 4 个段寄存器存放了 CPU 当前可以寻址的 4 个段的地址，也即可以从这四个段寄存器规定的逻辑段中存取指令代码和数据。如果 CPU 需要从其他的逻辑段存取指令代码或数据，那么程序必须首先改变对应段寄存器中的内容。CPU 中的 BIU 根据要访问的地址类型从不同的来源获得存储单元的逻辑地址，如表 2.3 所示。

表 2.3　逻辑地址的来源

操 作 类 型	正常使用段	可 替 换 段	偏 移 地 址
取指令	CS	无	IP
堆栈操作	SS	无	SP
变量（以下情况除外）	DS	CS、ES、SS	有效地址 EA
源数据串	DS	CS、ES、SS	SI
目的数据串	ES	无	DI
BP 作基地址寄存器	SS	CS、ES、DS	有效地址 EA

指令代码总是从当前的指令代码段中取得的，IP 中存有目标指令相对于这一段起点的偏移量。堆栈操作指令总是对当前堆栈段进行操作，SP 中存有栈顶的偏移量。

段的位置不受任何约束，只要段起始物理地址低 4 位为 0 即可，段与段的关系可以是首尾相连，也可以完全分离，或者部分重叠或完全重叠。一般情况下，各段在存储器中的分配是由操作系统负责的。存储器分段结构有利于使用模块化软件设计技术，同时也允许编写与位置无关的程序，即可动态浮动的程序。这使得多道程序系统或多重任务系统能充分使用现有的存储器容量。

2.2.3　堆栈

堆栈是微型计算机中很重要的一个存储区，用它可暂存一些在某些特殊程序运行中需要保护的数据或地址。堆栈最典型的用途是在调用子程序时，为实现子程序的正确返回，将断点（也即调用位置）的地址和调用程序中的一些数据暂存起来。存储数据时，与 8086/8088 指令队列不同，指令队列采用的是“先进先出”的原则，而堆栈采用的数据存储方式类似于堆放货物，按“后进先出”或“先进后出”的原则进行数据的存取。

IBM-PC 计算机系统采用了存储器分段方式，为了访问堆栈这部分存储区域，使用堆栈段来表示。堆栈段中存取数据的地址由堆栈段寄存器 SS 和堆栈指针 SP 来指示，SS 段寄存器中存放的是堆栈段的首地址，SP 中则存放栈顶的位置，此地址表示的是堆栈栈顶离段首地址的偏移量，数据的存取操作均在栈顶进行。堆栈段的示意图如图 2.7 所示。

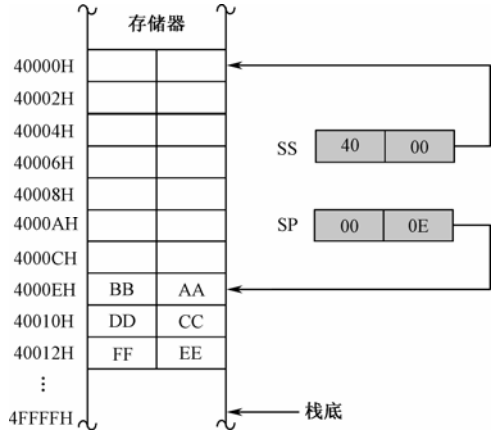


图 2.7　8086 系统的堆栈

2.3 Intel 80x86系列微处理器简介

2.3.1 80386 微处理器

1. 80386 CPU的特点

1985 年 10 月 Intel 公司推出了高性能全 32 位微处理器 80386DX，它是 80X86 处理器中第 1 个 32 位微处理器，使用了更先进的集成工艺，内部集成了 27.5 万个晶体管，时钟频率从 12.5MHz 至后来的 33MHz。80386DX 在内存管理和处理速度上比 80286 之前的 CPU 有了很大的突破，是一种功能完善高可靠性的 CPU，并在目标代码上保持与 8086、80286 的向上兼容，还为后续的 80486 和 Pentium 等 32 位机奠定了基础，是 CPU 发展的一个里程碑，其主要特点如下：

（1）全 32 位结构，具有 32 位数据总线和地址总线，能灵活处理 8 位、16 位、32 位和 64 位数据类型；CPU 能直接输出 32 位的地址信息，可直接寻址的物理空间达 4GB，除了保留存储器分段管理外，还增加了内存分页管理。

（2）内部结构由总线接口单元、指令预取部件、指令译码部件、执行部件、分段部件和分页部件 6 个逻辑功能部件组成，6 个部件都能独立操作，又可以对同一指令的不同部分同时并行操作，使多条指令重叠进行，大大提高了 CPU 的速度。

（3）80386 可以按实地址、保护虚地址以及虚拟 8086 三种模式工作，前两种模式与 80286 相同，虚拟 8086 模式是 80386 新增加的一种模式，在这种方式下，每个任务都是用 8086 语义运行，从而可以运行 8086 的各种软件。

（4）80386 增加了可测试性和调试功能，可测试特性包括自测试和对页面转换高速缓存的直接访问。

（5）为增强浮点数运算能力，Intel 还推出了与之配套的浮点协处理器 80387。

1988 年 Intel 推出了介于 80286 和 80386DX 之间的一种芯片 80386SX，80386SX 的外部数据总线和地址总线都与 80286 相同，分别是 16 位和 24 位，寻址能力 16MB。1990 年 Intel 推出了 80386SL 和 80386DL 两种芯片，80386SL 是基于 80386SX，80386DL 是基于 80386DX。这两种芯片都是低功耗、节能型的，是为便携计算机和节能型台式机，增加了系统管理方式 SMM（System Management Mode），当进入 SMM 方式后，CPU 会自动降低运行速度，控制显示器和硬盘等其他部件暂停工作，进入“休眠“状态，以达到节能的目的。

2. 80386CPU寄存器结构

80386 寄存器组是 8086、80286 寄存器组的超集。即在以前寄存器组的基础上加以扩充而成。寄存器组共有 32 个寄存器分为以下 6 类。

（1）通用寄存器

80386 内部共有 8 个 32 位的通用寄存器，分别命名为：EAX，EBX，ECX，EDX，ESI，EDI，EBP 和 ESP，如图 2.8 所示。可以用来存放数据和地址，这些寄存器的低 16 位可以单独访问，命名为 AX、BX、CX、DX、SI、DI、BP 和 SP。8 位操作时，可以单独访问 AX、BX、CX 和 DX 的低位字节 AL、BL、CL 和 DL，也可以单独访问高位字节 AH、BH、CH 和 DH。这样的设置是为了和 16 位 CPU 8086、80286 保持兼容。

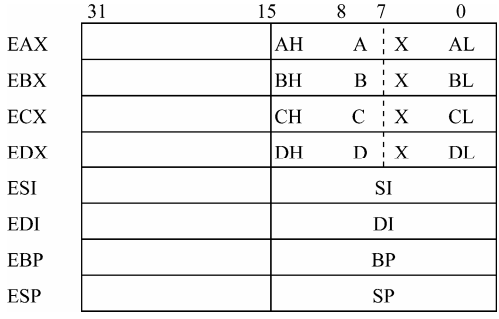


图 2.8　通用寄存器组

（2）状态寄存器

80386 状态寄存器包括 2 个 32 位的寄存器：指令指针 EIP 和标志寄存器 EFR。

EIP——用于保存下一条待取指令的偏移地址，EIP 的低 16 位称为 IP，与 8086 一样，用于标识 16 位指令代码的位置，这为在 80386 上执行 8086 或 80286 程序提供了可能。特别注意：若在 80386 上运行 16 位指令代码，使用 IP 作为偏移地址，其最大代码段为 64KB（2^{16}）；若运行 32 位指令，使用 EIP 作为偏移地址，其最大代码段为 4GB（2^{32}）。

EFR——（又称 EFlags）32 位标志寄存器，它的低 16 位与 80286 各位定义相同，在执行 8086 或 80286 指令代码时使用。EFR 格式如图 2.9 所示。其中 RF 位是恢复标志（与指令断点异常相关的标志位）。若 RF=1，即使遇到断点或调试故障，也不产生异常中断。VM 为虚拟 86 模式标志，若 VM=1，CPU 工作在虚拟 86 模式下。

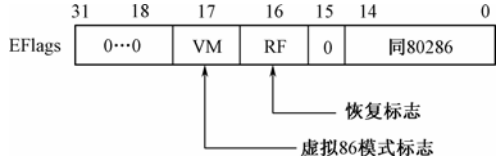


图 2.9　EFlags 寄存器新增标志

（3）段选择寄存器

80386 的段选择寄存器是在 80286 选择寄存器基础之上新增了 2 个支持当前数据段的段选择寄存器 FS 和 GS，与段选择寄存器相关联的段描述符高速缓冲器扩充到 64 位，如图 2.10 所示。

（4）系统地址寄存器

80386 有 4 个专用的寄存器用来保存保护模式下的表和段，这些寄存器命名为 GDTR，IDTR，LDTR 和 TR，它们对应地保存下面的表和段，即 GDT（全局描叙符表）、IDT（中断描叙符表）、LDT（局部描叙符表）、TSS（任务状态段），如图 2.11 所示。

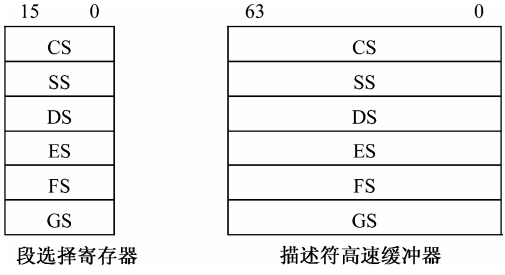


图 2.10　段选择寄存器

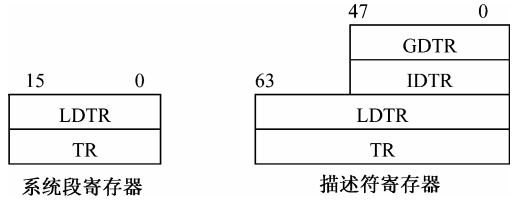


图 2.11　系统地址和系统段寄存器

（5）控制寄存器

80386 有 3 个 32 位的控制寄存器，CR0，CR2，CR3。它们用于保存全局性质的机器状

态。CR1 为 Intel 公司保留。

（6）调试寄存器和测试寄存器

80386 设置有 8 个调试寄存器，分别命名为 DR0～DR7，其中 DR0、DR1、DR2、DR3、DR6、DR7 供程序员进行程序调试，DR4、DR5 为 Intel 公司保留。

80386 设置有 2 个测试寄存器 TR6 和 TR7，用于存放需要测试的数据。

2.3.2 Pentium微处理器

1. Pentium CPU的特点

1993 年，Intel 推出了第 5 代的微处理器 Pentium。寄存器的长度为 32 位，包含有 64 位数据总线，32 位地址线。Pentium 处理器集成了 310 万个晶体管，最初推出的初始频率是 60MHz、66MHz，后来提升到 200MHz 以上，由于采用了很多新技术，指令系统更加丰富，规模更加庞大。

2007 年 11 月，Intel 公司公布了基于 45 纳米技术的全新处理器，处理器中增加了许多新的特性，如全新的 SIMD 流指令扩展 4(SSE4)，可通过 47 条全新指令加快包括视频编码在内的工作负载的处理速度，从而支持高清晰度画质和照片处理，以及重要的 HPC 和企业应用。同年，Intel 公司公布了 32 纳米技术并展示了 32 纳米的晶元，并宣布 2009 年将投产 32 纳米逻辑处理器。一个句点的面积就可以容纳超过 400 万枚晶体管，由此可见一枚 32 纳米晶体管的体积之小。

2. Pentium 处理器的基本结构

Pentium 处理器内部结构如图 2.12 所示。它由总线接口部件、代码高速缓冲存储器（代码 Cache）、数据高速缓冲存储器（数据 Cache）、转移目标缓冲器、控制 ROM 部件、控制部件、指令预取部件、指令译码部件、整数运算部件、浮点数运算部件、整数及浮点数寄存器组等功能部件组成。

（1）总线接口部件

总线接口部件负责与外部存储器和 I/O 接口设备进行数据交换，包括地址收发器和驱动器、数据总线收发器、总线宽度控制、写缓冲、Cache 控制、奇偶校验生成等。总线接口部件使用 32 位的地址总线和 64 位的数据总线与外部连接，可提供 4 种外部数据宽度，可以使用 64 位、32 位、16 位和 8 位宽度的数据总线。

（2）存储器管理部件

Pentium 处理器的存储器管理部件包括段式管理和页式管理两部分。Pentium 存储器虚拟地址被称为逻辑地址，其长度 48 位，由 16 位段地址和 32 位位移地址构成。Pentium 存储器采用段页式地址转换机制，程序对主存的调入和调出是按页面进行的，通过段地址查阅段表，将表中的地址与位移地址相加后得到 32 位的线性地址，然后通过页面转换得到物理地址。页面转换是通过页目录和页表实现，线性地址由页目录（10 位）、页号（10 位）和位移地址（12 位）组成，页面大小为 4KB。为了加快转换速度，通常需要设置 TLB 表。

（3）指令预取部件

在总线部件不执行其他总线周期时，指令预取部件执行一个取指令周期，预取 32 个字节的指令序列送到预取缓冲部件和 Cache 部件。当执行部件需要指令时，指令从预取部件传送到指令译码部件。预取部件的总线优先权最低，不影响其他的总线周期。

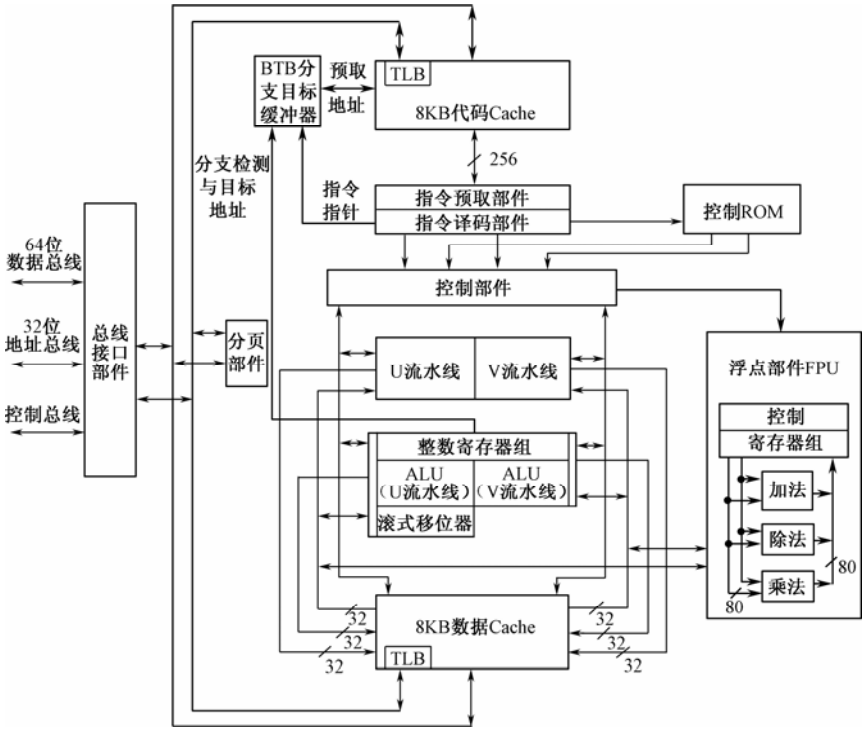


图 2.12　Pentium 处理器内部结构图

（4）整数流水线

Pentium 处理器内部集成了两条指令流水线，U 流水线和 V 流水线，每个流水线均有自己的 ALU、地址生成电路、数据 Cache 接口，可以在一个机器周期内发出两条整数指令，由于内部采用了超标量执行技术，允许两条指令以并行的方式执行。由于处理器内部采用了分支转移预测技术，出现转移时，转移目标缓冲器 BTB 可以保存 256 个转移预测，使转移预测失败的概率最小，当程序发生转移时，不因流水线的断流而影响性能。

（5）浮点流水线

浮点流水线由浮点接口、寄存器组及控制部件（FIRC）、浮点加法部件（FADD）、浮点乘法部件（FMUL）、浮点除法部件（FDIV）、浮点指数部件（FEXP）和浮点舍入部件（FRND）等部件构成。浮点流水线支持单精度（32 位）、双精度（64 位）、3 倍精度、80 位扩展精度浮点运算。

（6）Cache 部件

Pentium 处理器设置的片内 Cache 是分离的，分为指令 Cache 和数据 Cache，每个 Cache 容量 8KB，采用二路组相连的地址映像技术，Cache 行的长度为 32B。数据 Cache 有两个端口，分别与两个 ALU 交换数据，每个端口传送 32 位数据，也可以组合成 64 位数据，与浮点部件接口相连，传送浮点数。Pentium 处理器支持“修改/排他/共享/无效”（modified/exclusive/shared/invalid，简称 MESI）协议，数据 Cache 的每一行包含两个状态位，每一 Cache 行处于上述 4 种状态之一，写入数据时，先查询 Cache 是否命中，如命中，则根据 Cache 行的状态进行相应的写入数据操作，并修改（或保留）原状态位。

（7）指令译码部件

Pentium 处理器采用两步流水线译码方案，可以在每一个机器周期提供一个译码完成的

指令。指令译码部件从控制 ROM 中读出微指令序列，启动相应的控制动作，同时还有一些硬布线的微指令，在微代码执行前就开始动作。

（8）控制部件

Pentium 处理器采用微指令和硬布线两种方式控制计算机各部件的工作。控制部件得到指令译码后，一部分由控制 ROM 中的微指令序列产生控制信号，一部分由硬布线逻辑直接产生控制信号。

3. Pentium 寄存器结构

Pentium 寄存器组是 80386、80486 寄存器组的超集。即在以前寄存器组的基础上加以扩充而成，其通用寄存器、段寄存器、系统地址寄存器、控制寄存器、调试寄存器与 80386 基本相同，并对标志寄存器进行了扩展，取消了测试寄存器，其功能由“模型专用寄存器”MSR（Model Special Registers）来实现。

（1）标志寄存器

在标志寄存器中，新增了 AC 位、VIF 位、VIP 位和 ID 位，如图 2.13 所示。

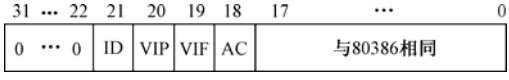


图 2.13　Pentium 标志寄存器

AC——（Alignment Check）地址对齐检查标志（80486SX 处理器有效）。若 AC=1 时进行地址对齐检查，当出现地址不对齐时会引起地址异常，只有在特权级 3 运行的应用程序才检查引起地址不对齐的故障。若 AC=0 时不进行地址对齐检查。该位主要是为与 80487SX 协处理器同步工作用的。

VIF——（Virtual Interrupt Flag）虚拟中断标志，是虚拟方式下中断标志位的映像。

VIP——（Virtual Interrupt Pending）虚拟中断挂起标志。与 VIF 配合，用于多任务环境下，给操作系统提供虚拟中断挂起信息。

ID——（Identification）标识标志。若 ID=1，则表示 Pentium 支持 CPU ID 指令。CPU ID 指令给系统提供 Pentium 微处理器有关版本号及制造商等信息。

（2）模型专用寄存器

Pentium 微处理器取消了测试寄存器 TR，由一组 MSR 替代，MSR 用于执行跟踪、性能检测、测试和机器检查错误。Pentium 微处理器采用两条指令 RDMSR（读 MSR）和 WRMSR（写 MSR）来访问这些寄存器，ECX 中的值确定将访问该组寄存器中哪一个 MSR。ECX 寄存器的值与 MSR 寄存器的关系如表 2.4 所示。

表 2.4　ECX 寄存器与 MSR 寄存器的关系

ECX	寄存器名称	说　　明
00H	机器检查地址	引起异常周期的存储地址
01H	机器检查类型	引起异常周期的存储周期类型
02H	测试寄存器 1	奇偶校验逆寄存器
03H	保留	
04H	测试寄存器 2	指令超高速缓冲存储器结束位
05H	测试寄存器 3	超高速缓冲存储器测试数据
06H	测试寄存器 4	超高速缓冲存储器测试标志
07H	测试寄存器 5	超高速缓冲存储器测试控制
08H	测试寄存器 6	TLB 测试线性地址

续表

ECX	寄存器名称	说　　明
09H	测试寄存器 7	TLB 测试控制和物理地址 31～12
0AH	测试寄存器 8	TLB 测试物理地址 35～32
0BH	测试寄存器 9	BTB 测试标志
0CH	测试寄存器 10	BTB 测试目标
0DH	测试寄存器 11	BTB 测试控制
0EH	测试寄存器 12	执行跟踪和转移预测
0FH	保留	
10H	时间错计数器	性能监测
11H	控制和时间选择	性能监测
12H	计数器 0	性能监测
13H	计数器 1	性能监测
14H	保留	

2.4　外部设备

外部设备是微型计算机系统的重要组成部分。计算机运行时的程序和数据都要通过输入设备送入机器，程序运行的结果要通过输出设备送给用户，所以输入、输出设备是计算机必不可少的组成部分。对于外部设备的管理也是汇编语言的重要使用场合。外部设备与主机之间的信息交换是通过接口进行的。

1. 为什么需要接口

因为计算机与外设进行数据交换时，所面对的外部设备的种类是多种多样的，可以是机械式、电子式或其他形式；输入的信息也是不相同的，可以是数字、模拟信号。不同设备的输入速度差异也相差很大，如键盘为秒级，而磁盘的速度则快得多。因此，需要一种部件能将 CPU 与外部设备的工作协调起来，有效完成输入、输出信息的任务，这种部件就是接口。

2. 接口的组成

每一个接口都包括一组寄存器和其他辅助电路。一般来说，这些寄存器有 3 种不同的用途。

（1）数据寄存器

用来保存在外部设备和 CPU 之间传递的数据，起数据缓冲或锁存的作用。

（2）状态寄存器

用来保存外部设备或接口的状态信息（如：忙/闲状态，准备就绪状态等），以便 CPU 在必要时测试外设的状态，了解外设的工作情况。

（3）控制寄存器

用来接收并存放 CPU 发出的各种控制命令（或控制字）及其他信息。这些控制命令的作用包括设置外设或接口的工作方式、工作速度、指定某些参数及功能、启停设备工作等。

3. 访问方式

为了方便 CPU 访问外部设备，给予外部设备中的每一个寄存器一个端口（PORT）地址

（又称为端口号），这样，这些端口地址的集合又构成了一个地址空间。CPU 访问时，使用专用的 IN 和 OUT 指令完成数据的传递。对于 IBM PC/XT 机的 I/O 地址空间可达 64KB，可有 64K 个字节端口或 32K 个字节端口，地址范围为 0000H～FFFFH。

对于用户使用外设进行数据传递，可以采用直接传送、查询中断传送或组传送等方式。在计算机系统中，用户最常用到的是两种例行程序的调用，一种为 BIOS（BASIC INPUT/OUTPUT SYSTEM），另一种为 DOS 功能调用。它们都是系统编制的子程序，通过中断方式转入所需要的子程序执行，执行完成之后返回原来的程序继续执行。这些例行程序有的完成一次简单的外设信息传送，如从键盘输入一个字符，或送一个字符至显示器等。有的完成相对复杂的一次外设操作，如从磁盘读/写一个文件等。总之，操作系统把一些复杂的外设操作编成例行程序，使用户用简单的中断指令（INT）就可以进入这些例行程序，完成所需的外设操作。

BIOS 和 DOS 功能调用虽然都是系统提供的例行程序。但是它们之间又有差别。BIOS 存放在机器的只读存储 ROM 中，它的层次比 DOS 更低，更接近硬件。DOS 功能调用是操作系统 DOS 的一个组成部分，在系统初始化后，常驻内存中，在它的例行程序中可以一次或多次调用 BIOS 以完成比 BIOS 更高级的功能，用户需要时，尽可能使用层次较高的 DOS 功能调用。但若不能满足程序需要时，就需要直接使用 BIOS，如果 BIOS 还不能完成，则需要自行编写中断处理程序。

习　　题

2.1　有两个 16 位的字节数据和一个 8 位字节数据，BFD2H、92B7H、39H，分别存放于存储器的 000D3H、000D7H 和 000D9H 单元中，请画图表示出它们在存储器的存放情况。

2.2　请将下列左边项和右边的解释联系起来，在括号中填入左边对应的数字。

（1）DF　　（　）A 进位标志
（2）AF　　（　）B 偶标志
（3）CF　　（　）C 方向标志
（4）ZF　　（　）D 陷阱标志
（5）IF　　（　）E 半进（借）位标志
（6）SF　　（　）F 零标志
（7）OF　　（　）G 符号标志
（8）TF　　（　）H 溢出标志
（9）PF　　（　）I 中断标志

2.3　段地址和偏移地址为 2001H、0011H 的存储单元的物理地址是什么？如果段地址和偏移地址是 101AH、1000H 和 5A00H、130CH 呢？

2.4　已知（SS）=1040H，（SP）=0012H，欲将（CS）=0C2FH，（IP）=004BH，（AX）=0E3AH（SI）=4B7AH 依次入栈保存。

（1）试画出椎栈有效示意图。
（2）写出入栈后 SS 和 SP 的值。

2.5　请将左边的术语与右边的定义联系起来，在括号中填入右边所示的字母。

（1）逻辑地址　　（）A 由 8 位二进制组成的通用基本单元。

（2）访存空间　（）B 以后进先出方式工作的存储空间。

（3）字节　（）C 唯一能代表存储空间的每个字节单元的地址，用 5 位 16 进制表示。

（4）堆栈　（）D 由段基地址和偏移地址两部分组成，均用 4 位 16 进制数表示。

（5）段寄存器　（）E 指示下一条要执行的指令的地址。

（6）物理地址　（）F CPU 组成系统所能访问的存储单元总数。

（7）状态标志　（）G 保存各逻辑段的起始地址的寄存器，PC 机。寄存器有：CS，DS，SS，ES。

（8）IP　（）H 记录指令操作结果的标志，共六位。

2.6　有一个由 20 个字组成的存储区，其起始处段地址为 4701H，偏移地址为 2012H。试写出该存储区首单元和末单元的物理地址。

2.7　Pentium 处理器主要由哪几个功能部件组成？

2.8　Pentium 处理器的 Eflags 寄存器比 8086 处理器的 Flags 寄存器增加哪些功能位？

2.9　请指出接口的用途是什么。

第3章　指 令 系 统

本章以 8086/8088 指令系统为基础，从介绍指令的基本格式，操作数的寻址方式入手，讲述指令系统中各类指令的功能、使用方法和应用场合，并根据处理器的发展历程，讲述 80386 和 Pentium 处理器新增的指令。

3.1　指令格式

计算机的指令系统是指微处理器所能执行的各种指令的集合。微处理器的主要功能必须通过它的指令系统来实现。不同的微处理器有着不同的指令系统，其中每条指令与微处理器的一种基本操作相对应，这在设计微处理器时就已确定。

8086/8088 指令系统包括上千条指令，这里所谓的指令可分为机器指令和汇编指令。机器指令用二进制代码表示，便于计算机识别，但不利于用户记忆和交流；汇编指令则用一些简单的符号来表示机器指令，以弥补机器指令的不足。若无特殊说明，本书此后所论及的指令均为汇编指令。

微型计算机的每一条指令通常由两部分构成。

1. 操作码（OP-Code）

操作码指示计算机所要执行的操作类型。CPU 执行指令时，首先将操作码从指令队列送到执行部件 EU 中的控制单元，经指令译码器分析识别后，产生执行本指令操作所需的时序控制信号，控制计算机完成规定的操作。

2. 操作数（Operand）

操作数指出指令执行操作所需的数据或操作结果存放的位置。有两个操作数的指令中一个操作数称为源操作数，另一个称为目的操作数。源操作数和目的操作数是参加操作的两个操作数，而目的操作数又存放操作的结果，也就是说，运算后，参加运算的一个操作数将会丢失，但一般情况下不关心这个问题。如果以后的运算中还会用到这个操作数的话，则应在运算之前将其送到其他位置。

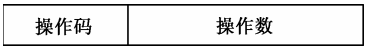


图 3.1　指令的基本格式

指令的基本格式如图 3.1 所示。

3.2　寻址方式

计算机是靠指令加工处理信息的，信息寄存在寄存器中或存储在存储器中，执行指令时往往要从寄存器或存储器中取出信息，加工处理后又存放到寄存器或存储器中。对于一条具体的指令，如何在寄存器或存储器中找到指令执行时所需的信息，如何确定执行结果的存放位置？这就需要了解寻址方式。寻址方式说明如何寻找操作数，以及如何确定执行结果存放

位置。8086/8088 指令涉及四种操作数：隐含操作数，立即操作数，寄存器操作数，以及存储器操作数。由此就有对应的 4 类寻址方式。

3.2.1 固定寻址（Inherent Addressing）

这种寻址方式下，操作数隐含在指令中。8086/8088 的某些单操作数指令规定操作数在 CPU 的某个固定的寄存器中，而这个寄存器又隐含在操作码中，例如：加法的十进制调整指令 DAA，其操作数总是固定隐含在 AL 中；还有的双操作数指令，例如寄存器的入栈、出栈指令只给出一个操作数，而另一个操作数被固定隐含在栈顶。

使用这种寻址方式的指令不需要计算存储器的有效地址 EA，执行速度较快。

3.2.2 立即寻址（Immediate Addressing）

这种寻址方式下，操作数以常量的形式出现在指令中。操作数随着指令一起进入指令队列，不必执行总线周期，故称为立即数。立即数可以为 8 位，也可以为 16 位。如果是 16 位数，则低字节存放在低地址单元中，高字节存放在高地址单元中。立即寻址方式用来表示常数，它常用于给寄存器赋初值，立即数只能用做源操作数，不能用做目的操作数。

【例 3.1】MOV　CX，9；立即数 9 作为源操作数赋给寄存器 CX。

【例 3.2】MOV　AX，5807H

指令执行后（AX）=5807H，如图 3.2 所示。图中指令存放在代码段中，OP 表示该指令的操作码部分，5807H 则为立即数，它是指令的一部分。

3.2.3 寄存器寻址（Register Addressing）

这种寻址方式下，操作数为通用寄存器或段寄存器（CS 除外）。16 位寄存器操作数可以是 AX、BX、CX、DX、SI、DI、SP、BP、DS、SS 及 ES；8 位寄存器操作数可以是 AL、AH、BL、BH、CL、CH、DL 和 DH。这种寻址方式由于操作数就是 CPU 的寄存器，不需要执行总线周期，执行速度较快。因此，在既可使用寄存器寻址又可使用后述的存储器寻址的场合，常常选用前者。

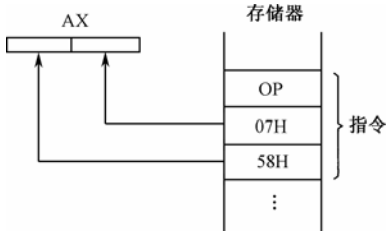


图 3.2　指令执行情况

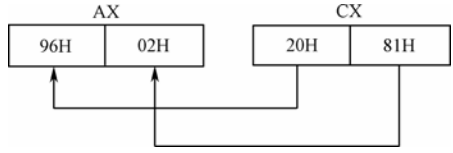


图 3.3　寄存器寻址方式执行情况

【例 3.3】MOV　AX，CX

若指令执行前（AX）=9602H，（CX）=2081H；则指令执行后，（AX）=2081H，（CX）内容保持不变。指令执行情况如图 3.3 所示。

3.2.4 存储器寻址

存储器寻址方式下，操作数一般是代码段之外的数据段、堆栈段、附加数据段中的存

储单元，指令给出的是存储单元的地址或产生存储单元地址的表达式。执行此类指令时，CPU 首先根据操作数字段提供的地址信息，由执行部件 EU 计算出有效地址 EA（EA 是一个不带符号的 16 位数据，代表操作数地址离段首地址的距离，即该地址到段首地址的字节数），再由总线执行部件 BIU 根据公式 PA=（16×段首地址）+EA 计算出操作数的物理地址。

一般而言，一条指令的目的操作数和源操作数不能同为存储器操作数。存储器寻址方式按 EA 计算方法的不同可以分为 5 种。

1. 直接寻址（Direct Addressing）

格式：（1）[常量]

（2）变量

直接寻址是最简单的存储器寻址。这种寻址方式下，操作数的有效地址由指令直接给出，是带有方括号的常量或是变量。

需要说明的是，该方式下，操作数的段地址默认为在数据段寄存器 DS 中，即 DS 为默认段寄存器。因此，直接寻址方式下，作为操作数的存储单元的物理地址为：

$$PA=16\times(DS)+nn$$

这里 nn 表示常量或变量的偏移地址。

【例 3.4】MOV　AL，[1000H]　　；将 DS 段 1000H 单元的内容传送到 AL 中。

【例 3.5】MOV　AX，[1000H]

与上一条指令不同的是，指令执行后将 DS 段中偏移地址为 1000H 的字单元内容传送到 AX，即低地址 1000H 对应 AL，高地址 1001H 对应 AH。若（DS）=2000H，（21000H）=3412H，则该指令执行后（AX）=3412H。指令执行情况如图 3.4 所示。

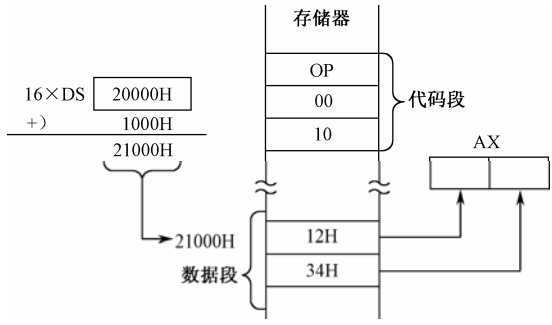


图 3.4　指令执行情况

使用直接寻址方式时应注意：

（1）直接地址可用数值表示，包括在[ ]之中，也可以用变量表示，例如：MOV AL，VALUE；这里，VALUE 为变量，变量是有属性的，它由数据段中定义数据的伪指令确定（伪指令将在第 4 章介绍）。

（2）若操作数在代码段、堆栈段或附加段中，则应在操作数地址之前使用前缀指出段寄存器名，这种前缀称为段超越前缀。例如：

MOV　AL，　ES：[2000H]　　；将 ES 段 2000H 单元的内容传送到 AL 中

段超越前缀也可以用于其他存储器寻址方式中，以使得给定的段寄存器取代默认的段寄存器。

（3）直接寻址方式适合于处理存储器的单个单元。IBM-PC 机中为了避免指令字的长度过长，规定双操作数指令除立即寻址方式之外必须有一个操作数使用寄存器或段寄存器，这就是一个变量常常先要送到寄存器中去的原因。

2. 寄存器间接寻址（Register Indirect Addressing）

格式：[BX、BP、SI 或 DI]

这种寻址方式下，操作数的有效地址 EA 不像直接寻址那样直接放在指令中，而是由基址寄存器 BX、BP 或变址寄存器 SI、DI 之一给出，即

$$EA=\begin{cases}(BX)\\(BP)\\(SI)\\(DI)\end{cases}$$

如果指令中使用的是 BX、SI 和 DI，则操作数在数据段中，且用数据段寄存器 DS 中的内容作为段地址，即操作数的物理地址为

$$PA=16\times(DS)+\begin{cases}(BX)\\(SI)\\(DI)\end{cases}$$

【例 3.6】MOV　AL，[BX]　　；设 BX 的内容为 1000H，则指令功能是将 DS 段 1000H 单元的内容传送到 AL 中。

【例 3.7】MOV　AX，[BX]

设（DS）=2000H，（BX）=1000H，（21000H）=3412H，物理地址 PA=16×2000H+1000H=20000+1000H=21000H，执行情况如图 3.5 所示。

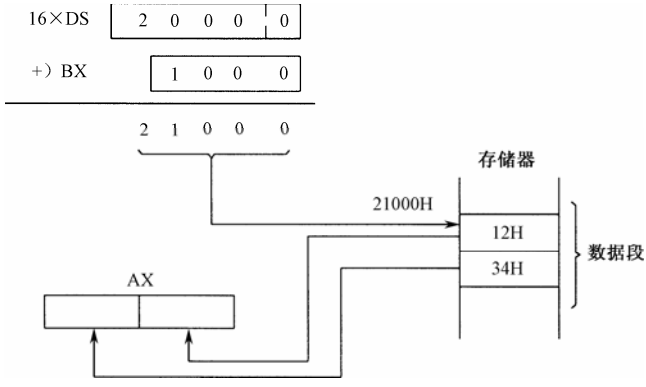


图 3.5　寄存器间接寻址执行情况

指令执行后，（AX）=3412H。

若指令中使用的是 BP，则操作数在堆栈段中，用堆栈段寄存器 SS 中的内容作为段地址，即操作数的物理地址为

PA=16×（SS）+（BP）

寄存器间接寻址通常用来对一维数组进行处理。只需在执行完一条指令后改变间接寻址寄存器 BX、BP、SI 和 DI 中的内容，就可以使用同一个地址表达式来指定一维数组中的不同元素，从而对连续的存储器单元进行存/取操作。

3. 基址寻址（Based Addressing）

格式：偏移量[BX 或 BP]

其中偏移量可以是常量或变量，还可以表示为[BX 或 BP][偏移量]和[BX 或 BP+偏移量]。

这种寻址方式与寄存器间接寻址方式的区别仅在于，只能使用基址寄存器 BX 和 BP，且指令中还要指定一个 8 位或者 16 位的偏移量，操作数的有效地址 EA 等于基址寄存器 BX 或 BP 的内容与偏移量之和，该偏移量在两个字节范围内，即

$$EA=\begin{cases}(BX)\\(BP)\end{cases}+\text{偏移量}$$

【例 3.8】MOV　AL，80H[BP]　　；设 BP 内容为 2040H，则指令功能是将堆栈段 20C0H 单元的内容传送到 AL 中。

该指令又可表达为

MOV　AL，[BP][80H]

或 MOV　AL，　[BP+80H]

【例 3.9】MOV　AX，COUNT[BX]

设（DS）=2000H，（BX）=1000H，COUNT=3000H，(24000H)=1058H，其寻址示意图如图 3.6 所示。指令执行后，（AX）=1058H。

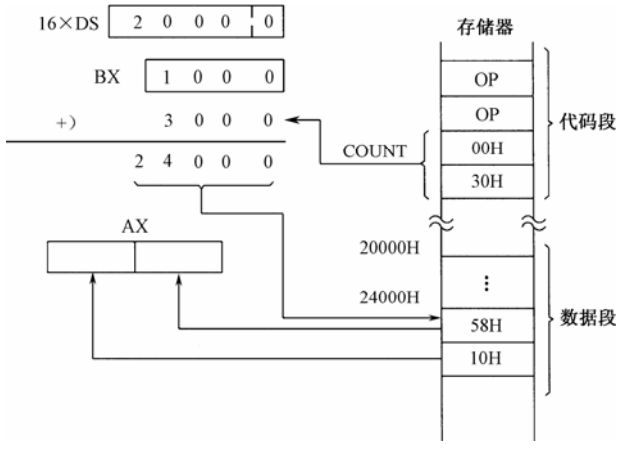


图 3.6　基址寻址方式执行情况

基址寻址通常也用来访问一维数组中的元素，用偏移量来确定数组的起点，基址寄存器的值选择一个元素。与寄存器间接寻址一样，因数组中所有元素具有相同的长度，只要改变基址寄存器的内容，就可以使用同一个地址表达式选择数组中任意的元素。

4. 变址寻址（Indexed Addressing）

格式：偏移量[SI 或 DI]。

其中偏移量可以是常量或变量，其他的表示形式类似于基址寻址方式，只需将基址寄存器 BX、BP 换成变址寄存器 SI、DI 即可。变址寻址中总是使用段寄存器 DS 的内容作段首地址，操作数的有效地址 EA 等于间址寄存器内容和位移量之和，即

$$EA=\begin{cases}(SI)\\(DI)\end{cases}+偏移量$$

变址寻址方式同样适合处理数组，通常 SI 用于源数组的变址寻址，DI 则用于目的数组的变址寻址。

例如：　MOV　AX，ARRAY1[SI]

　　MOV　ARRAY2[DI]，AX

其中 ARRAY1 和 ARRRAY2 分别用来表示源数组和目的数组的起点。若用上述两条指令，配上修改 SI、DI 值的指令，构成循环，就可实现将源数组移动到目的数组的目的。

5. 基址变址寻址（Based　Indexed　Addressing）

格式：偏移量[BX 或 BP+SI 或 DI]。

其中偏移量可以是常量或变量，其他的表示形式类似于基址寻址方式，只需将基址寄存器换成基址+变址寄存器即可。

这种寻址方式下，存储器操作数的有效地址 EA 是指令指定的基址寄存器 BX、BP 之一与变址寄存器 SI、DI 之一的内容以及偏移量三者之和，即

$$EA=\begin{cases}(BX)\\(BP)\end{cases}+\begin{cases}(SI)\\(DI)\end{cases}+偏移量$$

这里共有四种组合情况，并且根据基址是在 BX 还是在 BP 中，确定寻址操作数是在数据段还是堆栈段。对于前者，段寄存器使用 DS；对于后者，段寄存器使用 SS。基址变址寻址的操作数物理地址为

$$PA=16\times(DS)+(BX)+\begin{cases}(SI)\\(DI)\end{cases}+偏移量$$

和

$$PA=16\times(SS)+(BP)+\begin{cases}(SI)\\(DI)\end{cases}+偏移量$$

【例 3.10】MOV　AX，3000H[BX+SI]

设（DS）=1000H，（BX）=0400H，（SI）=1260H；

则：EA=0400H+1260H+3000H=4660H

　　PA=10000H+4660H=14660H

指令的执行情况如图 3.7 所示。指令执行后，（AX）=1058H。

应当注意以下指令是错误的：

　　MOV　AX，30H[BX][BP]

　　MOV　AX，10H[SI][DI]

基址变址寻址方式同样适合数组或表格的处理，由于基址和变址寄存器中的内容都可以改变，在处理二维数组时尤为方便。

这种寻址方式中，若偏移量为 0，则偏移量可默认。即指令

　　MOV　AX，0[BX+SI]

可表示为

　　MOV　AX，[BX+SI]

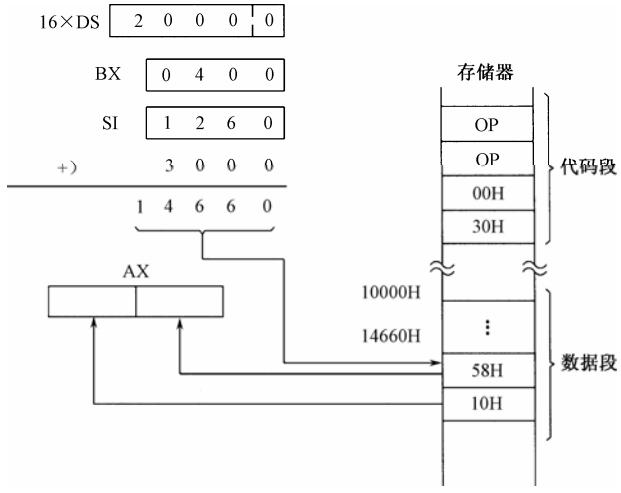


图 3.7　基址变址寻址方式执行情况

3.3　指令的执行时间

通常，一条指令的执行时间是指计算机取指令、取操作数、执行指令及传送结果各个阶段所需时间的总和。如果要详细讨论各种指令的执行时间是一个比较复杂的问题，这里只作简单介绍。

由于指令是存放在存储器中，因此运算器要执行指令就需先访问存储器，但是 8086/8088 CPU 的执行部件 EU 和总线接口部件 BIU 是并行工作的，BIU 可以预先把指令取到指令队列缓冲器中存放，形成了取指和执行的并行，这样，在计算指令的执行时间时，就不把取指时间计算在内。

执行指令的时间，除了 EU 中的基本执行时间外，有些指令在执行过程中可能需要多次访问内存，包括取操作数和存放操作数结果等，要执行总线的读/写周期，这样，执行一条指令的时间就是指令的基本执行时间及存取操作数时间的总和。指令的基本执行时间因指令的不同而异，相互之间有很大的差别，而存取操作数所需的计算有效地址 EA 的时间又随寻址方式的不同而异。

表 3.1 表示执行几种不同的指令所需的时间；表 3.2 表示执行加法时不同的寻址方式所需的时间；表 3.3 表示在不同的寻址方式下计算 EA 所需的时间。在这些表格中，所有的时间都以时钟周期数表示（计算机是按照节拍工作的，这里所说的节拍称为时钟周期）。

从表 3.1 至表 3.3 可以看出，不仅不同指令的执行时间差别很大，而且同一种指令使用不同的寻址方式时执行时间的差别也是很大的，通过以下例题可以对指令的执行时间有一个具体的了解。

【例 3.11】设 8086 的时钟频率为 5MHz（即时钟周期=0.2μs），试求两个字节相加的 ADD 指令在各种寻址方式下，指令的执行时间 t。

（1）目的操作数和源操作数均为寄存器操作数。需要 3 个时钟周期，即

t=3×0.2=0.6（μs）

（2）目的操作数为寄存器操作数，源操作数为基址变址寻址的存储器操作数。需要的时钟数为

t=9+EA=9+12=21（T）

第一项的 9 为这种寻址方式下指令的基本运算和基本操作时间，第二项为计算 EA 的时间，即

t=21×0.2（μs）=4.2（μs）

（3）目的操作数为基址变址寻址的存储器操作数，源操作数为寄存器操作数。需要的时钟数为

t=16+EA=16+12=28（T）=5.6（μs）

第一项的 16 为这种寻址方式下指令的基本运算和基本操作时间，第二项为计算 EA 的时间。

表 3.1　指令的基本执行时间举例

指　　令	寻 址 方 式	时钟周期数
加法 ADD	寄存器—寄存器	3
减法 SUB	寄存器—寄存器	3
传送 MOV	寄存器—寄存器	2
整数乘法 IMUL	16 位寄存器	128～154
整数除法 IDIV	16 位寄存器	165～184
移位	寄存器移位 1 位	2
调用	段内	19
无条件转移 JMP	直接	15
条件转移	满足条件	4
	不满足条件	16

表 3.2　加法指令的执行时间

操作数的寻址方式	时钟周期数	访问存储次数	指令长度 B
寄存器到寄存器	3	0	2
存储器到寄存器	9+EA	1	2～4
寄存器到存储器	16+EA	2	2～4
立即数到寄存器	4	0	3～4
立即数到存储器	17+EA	2	3～6

表 3.3　计算有效地址 EA 所需的时间

寻址方式	时钟周期数
直接	6
寄存器间接	5
基址	9
变址	9
基址变址	11～12

对于其他的寻址方式下 ADD 指令的执行时间，请读者练习计算。

从上述的例子可以看出：对于同一条 ADD 指令而言，因寻址方式不同，执行指令的时间也不同，即执行的效率有差异，有时这种差异还很大。从表 3.2 还可以得知，同一类的指令使用不同的寻址方式时指令的长度也不一样，占用存储器的字节数差异也很大。当一个实际的应用程序要求运行效率较高和占用空间较小时，程序的设计者不仅要研究算法、数据结构，还要研究指令与寻址方式的选用，才能编制出高效而简洁的程序。

3.4　Intel8086/8088指令系统

8086/8088 指令系统包括约百种指令助记符，它们与寻址方式结合，再加上操作数字节数的不同，可以构成上千种指令。这些指令按照功能可分为 6 类。

（1）数据传送指令；

（2）算术运算指令；
（3）位操作指令；
（4）串操作指令；
（5）转移指令；
（6）处理器控制指令。

本节将主要介绍前 3 类指令的格式和功能，后 3 类指令本节仅作概要介绍，详情将在后面的相关章节中讲解。为便于指令的介绍，现作以下约定。

（1）指令中的 IMM 表示立即数，IMM8 仅表示 8 位立即数，IMM16 仅表示 16 位立即数。

（2）指令中的 REG 表示寄存器操作数。它可代表 8 位寄存器 AH、AL、BH、BL、CH、CL、DH 和 DL，以及 16 位寄存器 AX、BX、CX、DX、SP、BP、SI 和 DI。REG8 仅表示 8 位寄存器，REG16 仅表示 16 位寄存器。指令中的 SEGREG 表示段寄存器操作数。它可代表 CS、DS、ES 及 SS。

（3）指令中的 MEM 表示存储器操作数，它可以表示任何一种存储器寻址方式下的操作数。MEM8、MEM16 及 MEM32 分别表示 8 位、16 位及 32 位存储器操作数。

（4）在对指令功能作说明时，用圆括号来表示所括起部分的内容。例如用（AX）表示 AX 的内容，用（2040H）表示偏移地址为 2040H 的存储单元的内容。

（5）在指令的图示中，对于存储器单元一般只标出偏移地址。

3.4.1 数据传送指令

数据传送指令用于寄存器、存储器或输入/输出端口之间传送数据或地址，这类指令共 14 条，按其特点可分为四组，如表 3.4 所示。

表 3.4 数据传送指令表

分 组	助 记 符	目的操作数	源 操 作 数	功 能
通用数据传送指令	MOV	REG/SEGREG/MEM	REG/SEGREG/MEM/IMM	传送
	PUSH	/	REG16/SEGREG/MEM16	入栈
	POP	REG16/SEGREG/MEM16	/	出栈
	XCHG	REG/MEM	REG/MEM	交换
累加器专用传送指令	XLAT	/	/	换码
	IN	AL/AX	输入端口	输入
	OUT	输出端口	AL/AX	输出
地址传送指令	LEA	REG16	MEM	传送 EA
	LDS	REG16	MEM32	传送段地址及 EA
	LES	REG16	MEM32	传送段地址及 EA
标志传送指令	LAHF	/	/	标志装入 AH
	SAHF			AH 送标志寄存器
	PUSHF			标志入栈
	POPF			标志出栈

1. 通用数据传送指令

（1）MOV 目的操作数，源操作数

把源操作数传送到目的操作数。

【例 3.12】MOV　AX, BX　　; AX←（BX），即将 BX 的内容传送到 AX 中;

MOV　CL, 80H　　; CL←80H;

MOV　AL, [2000H]　　; AL←（2000H），即将 EA 为 2000H（段地址取 DS 的值）单元的值传送到 AL 中。

说明：

① 源操作数不变（一般带有源操作数的 8086/8088 指令都不改变源操作数的内容，本书对于例外指令将予以说明）。

② 立即数、段寄存器 CS 不能作为目的操作数。源操作数和目的操作数不能同为存储器操作数，如图 3.8 所示。

以下指令均是错误的：

MOV　64H, BL

MOV　CS, AX

MOV　[2000H], [BX]

③ 源操作数和目的操作数类型必须一致，即同为单字节数，或同为双字节数，当指令中只有一个操作数的类型明确时，另一个操作数被视为同一类型。可以用“BYTE PTR”或“WORD PTR”将一个存储器操作数定义为字节或字类型。当存储器操作数为字类型时，寄存器或立即数的高字节对应其高地址，低字节对应其低地址（这种对应关系亦适用于其他指令）。

④ MOV 指令的执行不影响标志寄存器。

【例 3.13】MOV　[2000H]，AX

将 AX 的内容传送到偏移地址为 2000H 的存储单元中。指令执行情况如图 3.9 所示。

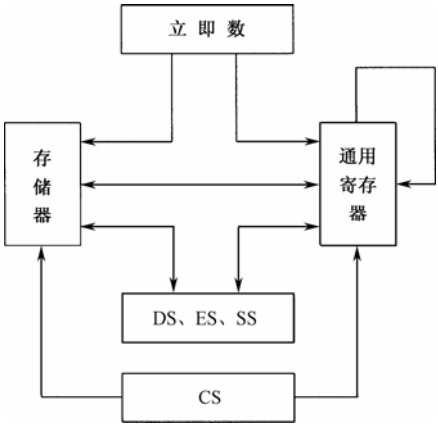


图 3.8　数据传送关系图

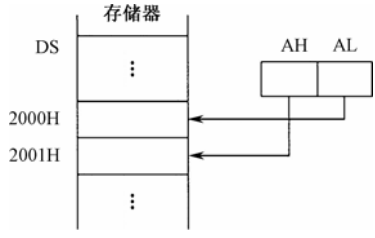


图 3.9　指令执行情况

【例 3.14】MOV WORD PTR [BX+SI]，80H

当（BX）=2080H，（SI）=1040H 时，表示将 0080H 传送到偏移地址为 30C0H 的存储单元中。指令执行情况如图 3.10 所示。

（2）PUSH　源操作数

将源操作数压入堆栈。先将 SP 的内容减 2，再将双字节的源操作数传送到 SP 所指示的堆栈栈顶。

【例 3.15】PUSH　BX　　；指令执行情况如图 3.11 所示。

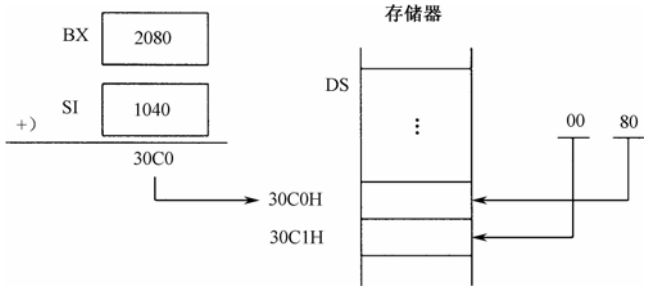


图 3.10　指令执行情况

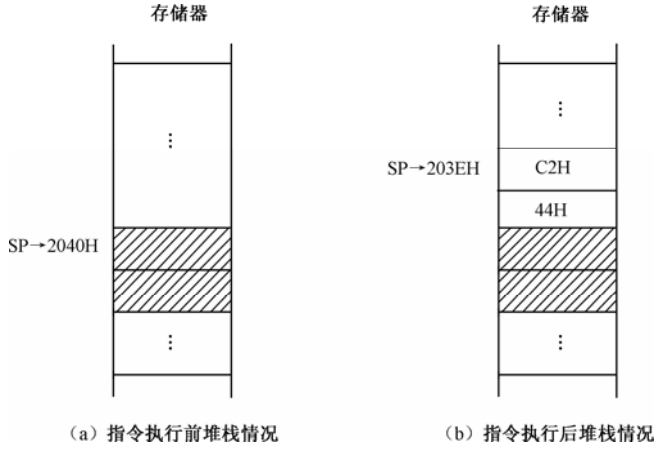


图 3.11　指令执行情况

（3）POP　目的操作数

将栈顶内容弹至目的操作数。先将 SP 所指单元也即堆栈栈顶的一个字传送到目的操作数，并将 SP 的内容加 2。

【例 3.16】POP　AX

若该指令紧接在上述 PUSH　BX 指令之后，则指令执行情况如图 3.12 所示。

堆栈是一个非常有用的存储结构，它遵循先进后出的原则，通常用于子程序的调用和返回，现场的保护和恢复等场合。PUSH 和 POP 指令一般应配对使用。使用时需注意以下 4 点：

① 堆栈操作指令中，有一个操作数是隐含的，这个操作数就是（SP）指示的栈顶存储单元。

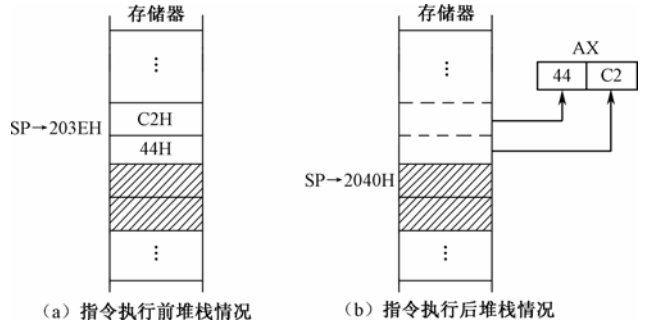


图 3.12　指令执行情况

② 8086/8088 堆栈操作都是字操作，不允许对字节操作。例如：PUSH　AH 不是正确指令。

③ 每执行一条入栈指令，（SP）自动减 2，高字节和低字节先后入栈；执行出栈指令时，则相反，低字节和高字节先后出栈，（SP）自动加 2。

④ CS 寄存器可以入栈，但不能随意弹出一个数据到 CS。

【例 3.17】设在一个被调用的子程序中，要使用到 AX、BX、CX 和 DX，子程序的运行还可能影响状态标志。为了使这些寄存器中的数据不被破坏，进入子程序时先予以入栈保护，子程序结束前再做出栈恢复。子程序中保护和恢复的程序段为

```
SUB1  PROC   NEAR          ; 定义过程
      PUSHF
      PUSH   AX
      PUSH   BX
      PUSH   CX
      PUSH   DX            ; 保护现场
      M
      POP    DX
      POP    CX
      POP    BX
      POP    AX
      POPF                 ; 恢复现场
SUB1  ENDP                 ; 过程结束
```

（4）XCHG　目的操作数，源操作数

把源操作数和目的操作数互换。说明：该指令与 MOV 指令功能上的区别有两点，其一，该指令不允许使用立即数和段寄存器作为操作数；其二，该指令改变源操作数的内容。

2. 累加器专用传送指令

（1）XLAT，把（BX）与（AL）相加形成有效地址 EA，将该单元中的单字节数传送到 AL 中。

【例 3.18】
```
MOV   BX, 4C02H
MOV   AL, 1DH
XLAT
```

有关存储单元情况如图 3.13 所示，则执行上述指令后，（AL）=55H。

这是一条专门用于 AL 和字节表中某一存储单元之间进行数据传送的指令。字节表的首地址在 BX 中，根据 AL 设置的偏移地址，就可以将该单元的内容传送到 AL 中。

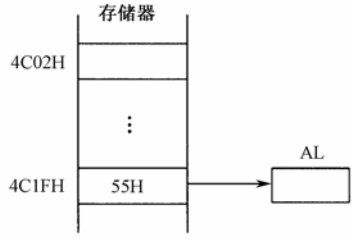


图 3.13　执行情况

（2）IN　累加器，端口地址

从指定端口输入一个字节到 AL 或输入一个字到 AX。端口地址以数值形式给出或通过 DX 间接给出。当端口地址大于 255 时，则只能由 DX 间接给出。

【例 3.19】IN　AX, 16H　　；从端口 16H 输入一个字到 AX 中。

【例 3.20】MOV　DX, 280H

IN　AL, DX　　；从端口 280H 输入一个字节到 AL。

说明：该指令及后述的 OUT 指令是专用于累加器和 I/O 端口之间进行数据传送的指令，操作数的确定方式有别于前述寻址方式。其一，端口地址不加“[ ]”；其二，使用 DX 作为专用间址寄存器。

（3）OUT 端口地址，累加器

实现输出，即与 IN 指令相反方向的数据传送。

3. 地址传送指令

地址传送指令实现操作数地址的传送。

（1）LEA　目的操作数，源操作数

将源操作数的有效地址 EA 传送给通用寄存器。

【例 3.21】MOV　BX, 0408H

MOV　SI, 2000H

LEA　BP, [BX+SI+6]　；将 240EH 送 BP（而不是将 240EH 单元的内容送 BP!）。

（2）LDS　目的操作数，源操作数

将源操作数指定的存储单元中的双字（通常为段地址和有效地址）传送给 DS 及目的操作数，高两字节送 DS，低两字节送目的操作数。

【例 3.22】已知（DS）=30C0H，相关存储区情况及执行以下指令的功能如图 3.14 所示。

LDS　SI，[40H]

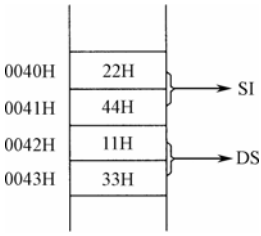


图 3.14　指令执行情况

（3）LES　目的操作数，源操作数

与 LDS 指令的区别仅在于，传送地址时将高两字节送 ES，而不是送 DS。

说明：

① 地址传送指令的源操作数必须是存储器操作数，目的操作数必须是 16 位通用寄存器。

② LEA 指令与 LDS、LES 指令所传送的有效地址有区别。LEA 指令所传送的有效地址为源操作数的有效地址，而 LDS 及 LES 指令所传送的有效地址在源操作数所指的存储单元中。

4. 标志传送指令

这组指令专用于对标志寄存器操作。如前所述，8086/8088 标志寄存器具有 16 位，LAHF 和 SAHF 仅对其低 8 位操作，而 PUSHF 和 POPF 对整个标志寄存器操作。

（1）LAHF，将标志寄存器低 8 位送 AH。

（2）SAHF，将（AH）送标志寄存器低 8 位。

（3）PUSHF，与 PUSH 指令功能相似，该指令的特殊之处仅在于，压入堆栈的是标志寄存器的内容。

（4）POPF，与 POP 指令功能相似，该指令的特殊之处仅在于，弹出的堆栈的内容是送标志寄存器。

标志传送指令中 SAHF 和 POPF 指令将直接影响标志寄存器的内容。利用这一特性，可以方便地改变标志寄存器中指定位的状态。

说明：数据传送指令中仅 SAHF 及 POPF 影响标志寄存器的内容。

3.4.2 算术运算指令

算术运算指令分为二进制数算术运算指令和 BCD 数算术运算调整指令。

1. 二进制数算术运算指令

参加算术运算的二进制数可以是单字节数或双字节数，也可以是无符号数或有符号数(有符号数在机器内部以补码形式表示)。由于汇编语言源程序中往往需要判断运算结果是否超出范围，是否为零，是否为负数等，所以二进制数算术运算指令除了产生与操作数位数相同的结果外，还将影响标志寄存器中的相应标志以便在必要时实现上述判断。该类指令共 14 条，按照四则运算分为 4 组，如表 3.5 所示。

表 3.5 二进制算术运算指令表

分组	助记符	目的操作数	源操作数	功能	对标志位的影响					
					OF	SF	ZF	AF	PF	CF
加法	ADD	REG/MEM	REG/MEM/IMM	相加	×	×	×	×	×	×
	ADC	REG/MEM	REG/MEM/IMM	带进位相加	×	×	×	×	×	×
	INC	REG/MEM	/	加 1	×	×	×	×	×	
减法	SUB	REG/MEM	REG/MEM/IMM	相减	×	×	×	×	×	×
	SBB	REG/MEM	REG/MEM/IMM	带借位相减	×	×	×	×	×	×
	DEC	REG/MEM	/	减 1	×	×	×	×	×	
	NEG	REG/MEM	/	取补	×	×	×	×	×	×
	CMP	REG/MEM	NEG/MEM/IMM	比较	×	×	×	×	×	×
乘法	MUL	/	REG/MEM	无符号数相乘	×	–	–	–	–	×
	IMUL	/	REG/MEM	有符号数相乘	×	–	–	–	–	×
除法	DIV	/	REG/MEM	无符号数相除	–	–	–	–	–	–
	IDIV	/	REG/MEM	有符号数相除	–	–	–	–	–	–
	CBW	/	/	字节转换为字						
	CWD	/	/	字转换为双字						

注：×表示根据操作结果设置标志；–表示标志不确定；空白表示标志不受影响。

（1）加法指令

① ADD 目的操作数，源操作数

将源操作数加到目的操作数，同时影响状态标志。

ADD 指令执行后对标志的影响：

- OF 字节运算结果超出字节有符号数的范围（−128～+127）或字运算结果超出字有符号数范围（−32768～+32767）时，OF=1；否则 OF=0。在把操作数视为有符数时，可通过该标志了解结果是否溢出。
- SF 运算结果的最高位为 1 时，SF=1；否则 SF=0（即 SF 与结果的最高位一致）。
- ZF 运算结果为零时，ZF=1；否则 ZF=0。
- AF 运算时，D_3 向 D_4 产生进位时 AF=1，否则 AF=0。
- PF 运算结果的二进制位 1 的个数为偶数时，PF=1；否则 PF=0。

• CF　运算时最高位产生进位时，即字节运算结果超出字节无符号数的范围（0～255），字运算结果超出字无符号数的范围（0～65535）时，CF=1；否则 CF=0。在把操作数视为无符号数时，可通过该标志了解结果是否溢出。

【例 3.23】ADD　AL，BL

设（AL）=0A4H，（BL）=5CH，则指令执行后，（AL）=0，OF=0，SF=0，ZF=1，AF=1，PF=1，CF=1。

编程者往往要根据不同情况关心不同的标志位。例如：执行该指令时，机器并不能判别所得数据是有符号数还是无符号数，而编程者在使用此指令及其他相关指令时总是为了解决某一个具体问题，从而对这一点是清楚的。因此，在使用此指令后，将会根据所得数据是有符号数还是无符号数来关心不同的标志位。

当把相关数据视为有符号数，从 OF=0 可知相加结果未溢出，也即 AL 的内容“0”就是两个有符号数 0A4H（−01011100B）和 5CH（+01011100B）之和。

当把相关数据视为无符号数，从 CF=1 可知相加结果产生进位也即“1 0000 0000B”就是两个无符号数 0A4H（10100100B）和 5CH（01011100B）之和。

说明：源操作数和目的操作数类型必须一致，即同为字节或同为字，且两者不能同为存储器操作数（这一点适用于所有双操作数的算术运算指令）。

② ADC　目的操作数，源操作数

功能与 ADD 指令基本相同，唯一区别是：将该指令执行前的 CF 值加至目的操作数中。该指令主要用于多字节加法运算。

【例 3.24】
```
MOV　DX,　2000H
MOV　AX,　8A04H
ADD　AX,　9D00H
ADC　DX,　45H
```

此程序段实现多字节数 20008A04H 与 459D00H 的相加。DX 存放有被加数的高两字节，AX 存放有被加数的低两字节。ADD 指令实现两数低两字节的相加，相加后（AX）=2704H，CF=1（相加结果超过两个字节）。ADC 指令实现两数高两字节的相加，且将 CF（即低两字节相加产生的进位）加至 DX，使 DX 内容为 2046H。

③ INC　目的操作数

功能与 ADD 指令基本相同，区别有两点：其一，隐含的源操作数为 1；其二，不影响 CF 标志。

该指令常用于某些计数器的计数，或用于修改地址。

【例 3.25】
```
MOV　SI,　2000H
MOV　AL,　[SI]
INC　SI
ADD　AL,　[SI]　　；（AL）为 2000H 单元和 2001H 单元内容之和
```

（2）减法指令

① SUB　目的操作数，源操作数

将目的操作数减去源操作数，同时产生相应标志。SUB 指令执行后对标志的影响与 ADD 指令基本相同，只不过这里只有“借位”而无“进位”。

② SBB　目的操作数，源操作数

功能与 SUB 指令基本相同，唯一区别是：目的操作数除减去源操作数外，还要减去该指令执行前的 CF 值。

该指令主要用于多字节减法运算。

③ DEC　目的操作数

功能与 SUB 指令基本相同，区别有两点：其一，隐含的源操作数为 1；其二，不影响 CF 标志。

④ NEG　目的操作数

对目的操作数取补码，结果送目的操作数。因为对一个数取补码相当于用 0 减去此数，所以该指令也属于减法运算指令。

⑤ CMP　目的操作数，源操作数

功能与 SUB 指令基本相同，唯一区别是：目的操作数不被差值取代。该指令的作用在于根据两操作数的大小关系产生状态标志值，以便其后指令根据状态标志确定程序流程。

（3）乘法指令

① MUL　源操作数

实现无符号数乘法运算。将（AL）或（AX）作为被乘数，源操作数作为乘数，乘积送 AX 或送 DX、AX。当源操作数为字节操作数时，默认（AL）为被乘数，乘积送 AX；当源操作数为字操作数时，默认（AX）为被乘数，乘积高两字节送 DX，低两字节送 AX。指令执行情况如图 3.15 所示。

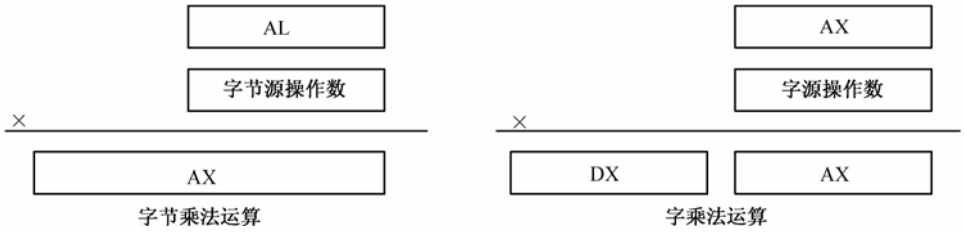


图 3.15　乘法指令执行情况

例如：

```
MUL   CL                    ; AL、CL 中的无符号数之积送 AX；
MUL   [SI]                  ; 此为错误指令。原因在于，计算机无法判别源操作数是字节操
                              作数抑或字操作数；
MUL   WORD  PTR  [SI]       ; AX 中的无符号数与 SI 所指单元的字无符号数相乘，乘积送
                              DX 和 AX。
```

MUL 指令影响进位标志 CF 和溢出标志 OF（其他标志不确定）。若乘积高半部分（字节相乘时乘积中的（AH）；字相乘时乘积中的（DX））非 0，则 CF=OF=1；否则，CF=OF=0。也即，CF=OF=1 标志着 AH 或 DX 中放有乘积的有效位。

② IMUL　源操作数

功能与 MUL 指令基本相同。区别仅在于，该指令实现有符号数乘法运算。此指令执行后，CF=OF=1 亦标志着 AH 或 DX 中放有乘积的有效位。即标志着（AH）或（DX）不是对应的低半部分的符号扩展。

【例 3.26】
```
MOV   AL, 0FCH          ; –4 送 AL;
MOV   CL, 1FH           ; +31 送 CL;
```

```
IMUL   CL              ;（AL）与（CL）之积–124 送 AX, 即（AL）=84H,（AH）
                       =FFH。此时 CF=OF=0, 标志着（AH）是（AL）的符号扩展。
```

（4）除法指令

① DIV　源操作数

实现无符号数除法运算。以（AX）或（DX）、（AX）中的内容为被除数，源操作数为除数。商送 AL 或 AX，余数送 AH 或 DX。当源操作数为字节操作数时，默认（AX）为被除数，商送 AL，余数送 AH；当源操作数为字操作数时，默认（DX）、（AX）内容为被除数，商送 AX，余数送 DX。指令执行情况如图 3.16 所示。

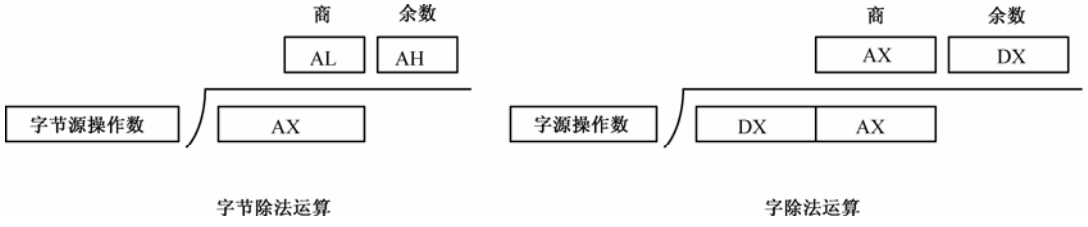


图 3.16　除法指令执行情况

```
【例 3.27】MOV   AX, 1001H     ; 4097 送 AX;
           MOV   CL, 20H       ; 32 送 CL;
           DIV   CL            ;（AX）与（CL）相除, 商 128 送 AL, 余数 1 送 AH。即（AL）
                               =80H,（AH）=01H。
```

DIV 指令执行后，各状态标志位不确定。

说明：当源操作数为字节类型时，商的范围为 0～255（FFH）；当源操作数为字类型时，商的范围为 0～65535（FFFFH）。超出范围则产生 0 号中断。

② IDIV　源操作数

功能与 DIV 指令基本相同，区别在于，该指令实现有符号数除法运算。另外，8086/8088 指令系统中规定余数的符号与被除数符号相同。例如：

```
MOV   CX, 4
IDIV  CX               ; 若已知 DX、AX 中放有 4001H, 则该指令执行后,（AX）=1000H,
                       （DX）=0001H。若已知 DX、AX 中放有–4001H, 则该指令执行后,
                       （AX）=F000H（即–1000H），（DX）=FFFFH（即–1）。
```

③ CBW

将（AL）的符号位扩展到 AH 中（即使得 AH 各位与 AL 最高位相同），该指令常用在 IDIV 指令之前。CBW 指令执行后，各状态标志位不确定。例如：

```
MOV   AL,  76H         ; +76H 送 AL;
CBW                    ; 0076H 送 AX, 即+0076H 送 AX。
```

又如：

```
MOV   AL, 98H          ; –68H 送 AL;
CBW                    ; FF98H 送 AX, 即–0068H 送 AX。
```

④ CWD

功能与 CBW 基本相同，区别仅在于，该指令是将（AX）的符号位扩展到 DX 中。

2. 十进制数算术运算调整指令

前面所述的算术运算都是针对二进制数，但人们最为常用的却是十进制数。在用计算机进行算术运算时，可以先将操作数做十→二进制转换，然后做二进制数算术运算，再将结果做二→十进制转换。为了便于十进制数的运算，8086/8088 系统还提供了一组十进制数算术运算调整指令，该类指令将在 5.2.1 中详细介绍。

3.4.3 位操作指令

8086/8088 提供的位操作指令包括逻辑运算指令和移位指令，这类指令可直接对寄存器或存储器操作数的位进行操作，如表 3.6 所示。

表 3.6 位操作指令表

分组	助记符	目的操作数	源操作数/计数	功能	对标志位的影响					
					OF	SF	ZF	AF	PF	CF
逻辑运算指令	NOT	REG/MEM	/	非						
	AND	REG/MEM	REG/MEM/IMM	与	0	×	×	–	×	0
	OR	REG/MEM	REG/MEM/IMM	或	0	×	×	–	×	0
	XOR	REG/MEM	REG/MEM/IMM	异或	0	×	×	–	×	0
	TEST	REG/MEM	REG/MEM/IMM	测试	0	×	×	–	×	0
移位指令	SAL/SHL	REG/MEM	1/CL	算术/逻辑左移	×	×	×	–	×	×
	SAR	REG/MEM	1/CL	算术右移	×	×	×	–	×	×
	SHR	REG/MEM	1/CL	逻辑右移	×	×	×	–	×	×
循环移位指令	ROL	REG/MEM	1/CL	循环左移	×					×
	ROR	REG/MEM	1/CL	循环右移	×					×
	RCL	REG/MEM	1/CL	带进位循环左移	×					×
	RCR	REG/MEM	1/CL	带进位循环右移	×					×

注：×表示根据操作结果设置标志；–表示标志不确定；空白表示标志不受影响。

1. 逻辑运算指令

（1）NOT 目的操作数

将目的操作数按位取反，结果送目的操作数。例如

```
MOV  AX, 0
NOT  AX          ; 使（AX）为 FFFFH。
```

（2）AND 目的操作数，源操作数

将目的操作数与源操作数按位相与，结果送目的操作数。当两个操作数的对应位都为 1 时，结果的对应位为 1；否则为 0。该指令常用于屏蔽目的操作的某些位，即使得目的操作数的某些位置 0，其余保持不变。例如

```
MOV  AL, 49H
AND  AL, 0FH     ; 使（AL）=09H, 即屏蔽（AL）的高 4 位, 而低 4 位不变。
```

（3）OR 目的操作数，源操作数

将目的操作数与源操作数按位相或，结果送目的操作数。当两个操作数的对应位都为 0 时，结果的对应位为 0；否则为 1。该指令常用于使目的操作数的某些位置 1，其余位保持不

变。例如

```
MOV   AL, 49H
OR    AL, 3CH      ; 使（AL）=7DH, 即使得（AL）中间 4 位置 1, 其余位保持不变。
```

（4）XOR　目的操作数，源操作数

将目的操作数与源操作数按位作异或操作，结果送目的操作数。当两个操作数的对应位不同时，结果的对应位为 1；否则为 0。该指令常用于判断两个数中哪些位不同，或用于改变指定位的状态。例如

```
MOV   DH,   12H
XOR   DH,   80H      ; 使（DH）=92H, 即改变 DH 最高位状态。
```

又如：XOR　AX,　AX　　；使（AX）=0。

（5）TEST　目的操作数，源操作数

功能与 AND 指令基本相同，唯一区别是：目的操作数保持不变。该指令常用于检测某种条件是否满足，但又不希望改变目的操作数的场合。例如

```
TEST  AL, 01H      ; 可用此指令检测 AL 最低位的状态。若 AL 最低位为 0, 则两操作数按位相
                     与的结果为 0, 从而 ZF=1; 若 AL 最低位为 1, 则结果为 1, 从而
                   ZF=0。
```

说明：逻辑运算指令中的两个操作数不能同为存储器操作数。

2. 移位指令

移位指令包括算术移位指令、逻辑移位指令和循环移位指令。这些指令只有目的操作数而无源操作数，并在指令中给出移位的位数。而且此位数只能用 1 或 CL 表示。

（1）算术、逻辑移位指令

算术移位指令（SAL、SAR）用于有符号数，而逻辑移位指令（SHL、SHR）用于无符号数。操作数的左移意味着小数点相对右移，而操作数的右移意味着小数点相对左移。算术、逻辑移位指令的功能如图 3.17 所示。

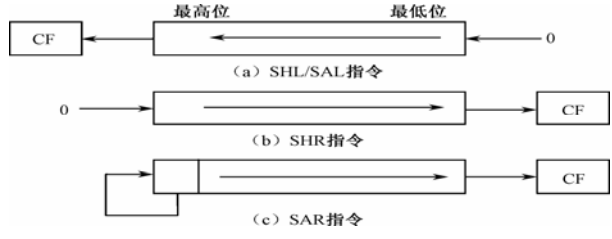


图 3.17

① SHL/SAL　目的操作数，计数

SHL 指令和 SAL 指令功能完全相同。将目的操作数左移若干位，每左移一位，最低位补 0，最高位送 CF。该指令可以方便地实现有符号数和无符号数乘以 2^n 的运算（n 为移位计数值），不过在使用时要注意是否发生溢出。例如

```
MOV   AL , 16H
SHL   AL ,  1      ; 使（AL） 即 0001 0110B 左移 1 位。移位后, （AL）=0010 1100B,
                     是原值的 2 倍。CF 值为 0。
```

又如：MOV　BH , 0EEH

```
MOV   CL,  2
SAL   BH , CL   ; 使 BH 中的有符号数 1110 1110B 左移 2 位。移位后（BH）=10111000B, 是
                  原值的 4 倍（原值为–0001 0010B, 即–18; 移位后的值为–0100 1000, 即–72）。
```

② SHR　目的操作数，计数

将目的操作数右移若干位。每右移一位，高位均补 0，最低位送 CF。该指令可以方便地实现无符号数除以 2^n 的运算。

③ SAR　目的操作数，计数

将目的操作数右移若干位。每右移一位，高位均保持不变，最低位送 CF。该指令可以方便地实现有符号数除以 2^n 的运算。例如

```
MOV   AL,  0F8H
SAR   AL,  1      ; 使 AL 中的有符号数 1111 1000B 右移 1 位。移位后（AL）=1111 1100B, 是
                    原值的 1/2（原值为–0000 1000B, 即–8; 移位后的值为–0000 0100B, 即–4）。
```

（2）循环移位指令

循环移位指令的功能如图 3.18 所示。

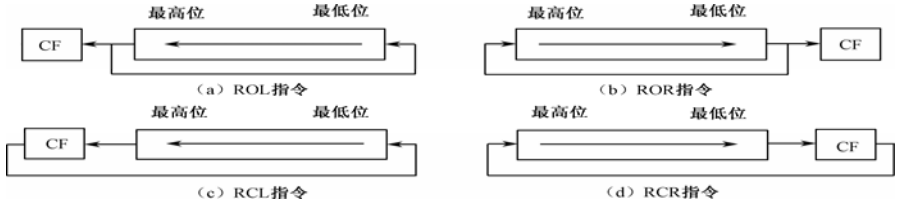


图 3.18

① ROL　目的操作数，计数

将目的操作数循环左移若干位。每左移一位，左移前的最高位送最低位以及 CF。

② ROR　目的操作数，计数

将目的操作数循环右移若干位。每右移一位，右移前的最低位送最高位以及 CF。

③ RCL　目的操作数，计数

将目的操作数连同 CF 循环左移若干位。每左移一位，左移前的最高位送 CF，左移前的 CF 送最低位。

④ RCR　目的操作数，计数

将目的操作数连同 CF 循环右移若干位。每右移一位，右移前的 CF 送最高位，右移前的最低位送 CF。

【例 3.28】

```
MOV   DX,  0FFF9H
XOR   AX,  AX          ; 使 AX, CF 清 0。
SAR   DX,  1
RCR   AX,  1
SAR   DX,  1
RCR   AX,  1
```

该程序段实现 DX、AX 中的有符号数 FFF9 0000H 右移两位。移位后 DX、AX 内容为 FFFE 4000H，是原值的 1/4（原值为–0000 0000 0000 0111 0000 0000 0000 0000B，移位后的值为–0000 0000 0000 0001 1100 0000 0000 0000B）。

3.4.4 串操作指令

串是指存储器中的字节串（字节序列）或字串（字序列）。8086/8088 系统对串的操作提供了 5 种基本的指令以及与之配合使用的重复前缀。它们常用于循环结构，本书将在 7.2 中对此做介绍。

3.4.5 转移指令

一般情况下指令是顺序地逐条执行的。但有时需要改变这种执行流程，8086/8088 系统为此提供了转移指令。转移指令用于分支结构，本书将在 6.2 中对此做介绍。

3.4.6 处理器控制指令

处理器控制指令用于控制 CPU 的某些功能，如表 3.7 所示。

表 3.7 处理器控制指令表

分 组	助 记 符	目的操作数	源 操 作 数	功 能
标志操作指令	STC	/	/	CF 置 1
	CLC	/	/	CF 置 0
	CMC	/	/	CF 取反
	STD	/	/	DF 置 1
	CLD	/	/	DF 置 0
	STI	/	/	IF 置 1
	CLI	/	/	IF 置 0
外同步指令	HLT	/	/	暂停，等待中断或复位
	WAIT	/	/	等待
	ESC	IMM	REG/MEM	交权
	LOCK	/	/	封锁总路线
空操作	NOP	/	/	空操作

3.5 Intel 80x86及Pentium指令系统

3.5.1 Intel80386 新增和扩充指令

80386 是 80X86 微处理器系列发展中的里程碑。80386 指令系统包括了所有 80286 指令，并对 80286 的部分指令进行了功能扩充，还新增了一些指令，特别指出的是，80386 提供了 32 位寻址方式可对 32 位数据直接操作。所有 16 位指令均可扩充为 32 位指令。80386 有 8 个 32 位通用寄存器：EAX，ECX，EDX，EBX，ESP，EBP，ESI，EDI。它们分别是原来的 16 位通用寄存器 AX，CX，DX，BX，SP，BP，SI，DI 的扩展（详细的寄存器结构在第 2.3 节已介绍），这些 32 位通用寄存器的低 16 位也可以作为 16 位通用寄存器独立存取数据，也就是说 80386 对 8086，80286 是向上兼容的。

对于数据段寄存器，80386 在原有基础上增加了两个：FS 和 GS。80386 的标志寄存器扩展到了 32 位，其中某些位没有定义。80386 在实地址模式下有 9 个标志位可用，在保护虚地址模式下有 13 个标志位可用，扩展后的标志寄存器也可称为 E 标志寄存器（EFR）EFLAGS。

80386 有实地址模式、保护虚地址模式和虚拟 8086 模式 3 种工作方式，在 DOS 环境中只能运行于实地址模式，可做为超高速 8086 芯片使用。

1. 数据传送与扩展指令

（1）MOVSX　寄存器，寄存器/存储器

将源操作数传送到目的操作数中。目的操作可以是 16 位或 32 位寄存器；源操作数可以是寄存器或存储器操作数，其位数应小于或等于目的操作数的位数。当源操作数的位数少于目的操作数时，目的操作数的高位用源操作数的符号位填补。此指令适用于有符号数的传送与扩展。该指令不影响状态标志位。

（2）MOVZX　寄存器，寄存器/存储器

与 MOVSX 功能基本相同，唯一区别在于，当源操作数的位数少于目的操作数位数时，目的操作数的高位用“0”填补。该指令适用于无符号数的传送与扩展。例如

```
MOV    DL, 86H
MOVSX    AX , DL                ; 86H 扩展成 FF86H 送 AX;
MOVSX    ECX, DL                ; 86H 扩展成 FFFF FF86H 送 ECX;
MOVZX    BX, DL                 ; 86H 扩展成 0086H 送 BX;
```

又如：

```
MOV    WORD    PTR [BX], 68H    ; 0068H 送 BX 所指定的内存单元;
MOV    AX, [BX]                 ; 0068H 送 AX;
MOVSX    ESI, WORD    PTR [BX]  ; 将 0068H 扩展成 0000 0068H 送 ESI;
MOVZX    EDI, WORD    PTR [BX]  ; 将 0068H 扩展成 0000 0068H 送 EDI。
```

2. 堆栈操作指令

（1）PUSH　8/16 /32 位立即数

将 8/16/32 位立即数压入堆栈。当然该指令执行后 SP 的值将减 2 或者 4。通常使用以下方法来区别操作数是 8 位，16 位还是 32 位立即数。

【例 3.29】
```
PUSH   'A'     ; 将 0041H 压入堆栈（8 位立即数）
PUSHW   15H    ; 将 0015H 压入堆栈（16 位立即数）
PUSHD   20H    ; 将 0000 0020H 压入堆栈（32 位立即数）
```

在 8086/8088 指令系统中，PUSH 指令允许的操作数只能是两字节的寄存器操作数或两字节的存储器操作数。该指令中如果给出的数不够 16 或 32 位，则自动扩展为 16 或 32 位后压入堆栈。

（2）PUSHA

将所有通用寄存器 AX，CX，DX，BX，SP，BP，SI，DI 的内容按顺序压入堆栈，入栈的 SP 值是执行该指令之前 SP 的值。在执行完本指令后，SP 值减 16，如图 3.19 所示。

（3）PUSHAD

将所有通用寄存器 EAX，ECX，EDX，EBX，ESP，EBP，ESI，EDI 的内容顺序压入堆栈，其中压入堆栈的 ESP 是该指令执行前 ESP 的值。执行该指令后，ESP 值减 32。

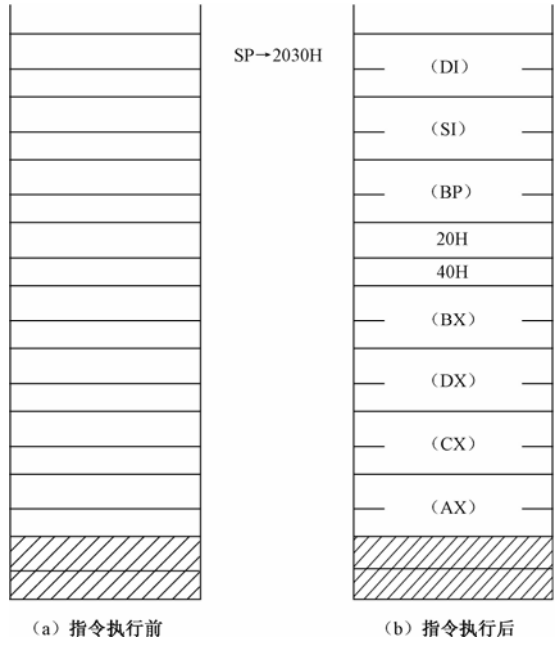


图 3.19　执行 PUSHA 指令的堆栈情况

（4）POPA

将栈顶的内容顺序弹至 DI，SI，BP，SP，BX，DX，CX，AX（次序与 PUSHA 指令相反）。SP 中的值是堆栈中所有通用寄存器弹出后，堆栈指针实际指向的值（不是栈中保存的 SP 值），也即该指令执行后 SP 的值，可以通过加 16 来恢复，如图 3.20 所示。PUSHA 及 POPA 均不影响状态标志位。

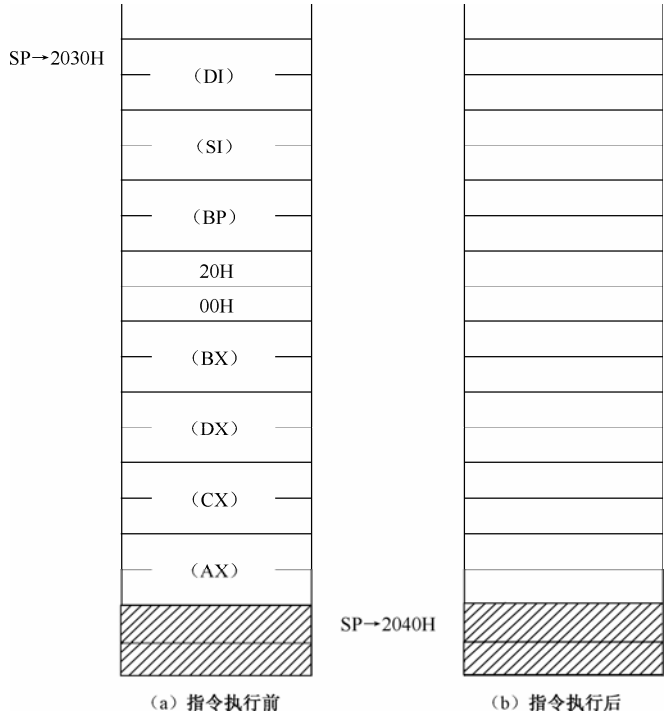


图 3.20　执行 POPA 指令的堆栈情况

【例 3.30】一个子程序被调用时，保存所有通用寄存器，可用下述指令实现：

```
PUSHA
CALL   SUB1
POPA
```

显然使用上述指令比使用 PUSH 指令和 POP 指令更方便。

（5）POPAD

将当前栈顶内容顺序弹至 EDI，ESI，EBP，ESP，EBX，EDX，ECX，EAX（次序与 PUSHAD 指令相反），但是最终 ESP 的值为弹出操作对堆栈指针调整后的值（而不是堆栈中保存的 ESP 的值）。也即执行该指令后 ESP 的值，可以通过增加 32 来恢复。

（6）PUSHFD

将 32 位标志寄存器 EFLAGS 的内容压入堆栈。

（7）POPFD

将当前栈顶的 4 字节内容弹至 EFLAGS 寄存器。

上述堆栈操作指令中，除 POPFD 以外，其余均不影响状态标志位。

（8）设置堆栈空间指令

格式：ENTER　16 位立即数，8 位立即数。

ENTER 指令使用两个操作数，16 位立即数表示堆栈空间的大小，也即表示给当前过程分配多少字节的堆栈空间，8 位立即数指出在高级语言内（如 Pascal 语言）调用自身的次数，也即嵌套层数。

说明：该指令使用 BP 寄存器而非 SP 作为栈基值。

【例 3.31】ENTER　6，0

该指令为过程分配了 6 个字节的堆栈空间，其嵌套层数为 0。指令执行情况如图 3.21 所示。

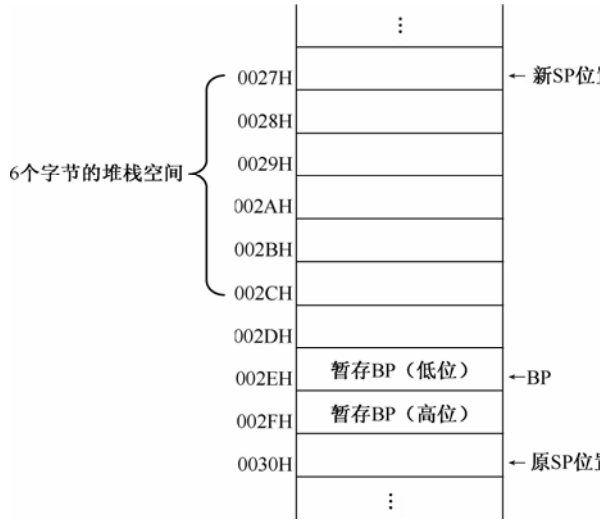


图 3.21　ENTER 指令执行情况

（9）撤销堆栈空间指令

格式：LEAVE

该指令无操作数，撤销由 ENTER 指令建立的堆栈空间。

【例 3.32】有一个 16 位参数，在主程序中需交给一个子程序处理，结果再传回主程序，可使用堆栈空间指令完成上述工作。

```
; 调用子程序的过程，参数存入到堆栈中
ENTER   4, 0                    ; 建立 4 字节堆栈空间
MOV     AX, Number1             ; 保存参数 1
MOV     [BP−4], AX
MOV     AX, Number2             ; 保存参数 2
MOV     [BP−2], AX
CALL    COURSE                  ; 调用子程序
MOV     AX, [BP−4]              ; 取结果 1
MOV     Number1, AX
MOV     AX, [BP−2]              ; 取结果 2
MOV     Number2, AX
LEAVE                           ; 撤销堆栈空间
    M
; 使用堆栈处理数据的子程序
COURSE      PROC    NEAR
PUSHA
MOV    AX, [BP−4]               ; 取参数 1
MOV    DX, [BP−2]               ; 取参数 2
    M
; 参数处理
    M
MOV    [BP−4], AX               ; 保存结果 1
MOV    [BP−2], DX               ; 保存结果 2
POPA
RET
COURSE      END
```

3. 地址传送指令

（1）LFS　寄存器，存储器

将源操作数所指存储单元的 4 字节或 6 字节内容送指定的寄存器及段寄存器 FS（FS 及 GS 为 80386 新增的段寄存器）。

当目的操作数为 16 位寄存器时，将 4 字节的存储器操作数中的低两字节送指定寄存器，高两字节送段寄存器 FS；当目的操作数为 32 位寄存器时，将 6 字节的存储器操作数中的低 4 字节送指定寄存器，高两字节送段寄存器 FS。该指令不影响状态标志位。

【例 3.33】LFS　BX, ARRAY

指令执行情况如图 3.22 所示。

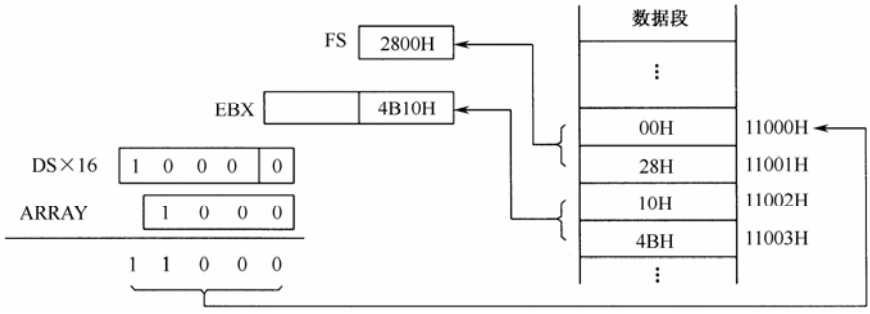


图 3.22　指令执行情况

（2）LGS　寄存器，存储器

该指令与 LFS 指令功能基本相同，唯一区别在于，该指令所涉及的段寄存器为 GS。

（3）LSS　寄存器，存储器

该指令与 LFS 指令功能基本相同，唯一区别在于，该指令所涉及的段寄存器为 SS。

4. 运算指令

在 8086/8088 指令系统中，乘法指令只给出一个操作数，另一个操作数隐含在 AL 或 AX 中，80386 将其扩充为可以有两个或 3 个操作数。

（1）IMUL　寄存器，寄存器/存储器/立即数

将通用寄存器中的有符号数作为被乘数，相同位数通用寄存器、存储单元中的有符号数或立即数（如立即数与目的操作数不等长，运算时机器会自动将其符号扩展成与目的操作数等长）作为乘数，乘积送目的操作数。若乘积溢出，溢出位部分将丢失，且将 OF 及 CF 置 1；否则将 OF 及 CF 置 0。

说明：目的操作数的位数必须与源操作数位数相同。

例如：　　IMUL　DX, 9

（2）IMUL　寄存器，寄存器/存储器，立即数

与前一指令功能基本相同，唯一区别在于，寄存器/存储器为被乘数，立即数为乘数，乘积存放在第一个操作数中。例如：

```
IMUL   EAX, DWORD DTR [BX], 9   ; 将 BX 所指定的 4 字节存储器操作数乘以 9, 乘积送 32 位
                                  通用寄存器 EAX。
```

（3）CWDE

将 AX 中 16 位有符号数的符号位扩展到 EAX 的高 16 位中，即把 AX 的 16 位有符号数扩展为 32 位后，送 EAX。

（4）CDQ

将 EAX 中 32 位有符号数扩展到 EDX：EAX 寄存器对中，使之成为 64 位有符号数，即将 EAX 中的符号位扩展到 EDX 中。

【例 3.34】若 DATA1 中为 16 位有符号数，值为−5，DATA2 中为 32 位有符号数，值为−7。将 DATA1 扩展成 32 位有符号数，将 DATA2 扩展成 64 位有符号数。

```
MOV   AX，DATA1          ;AX 中为–5（FFFBH）
CWDE                     ; 扩展后 EAX 中为 32 位的–5（FFFF FFFBH）
```

```
MOV   EAX，DATA2          ; EAX 中为–7（FFFF FFF9H）
CDQ                       ; 扩展后 EDX：EAX 中为 64 位的–7（FFFF FFFF FFFF FFF9H）
```

5. 移位指令

（1）移位指令助记符　寄存器/存储器　立即数（≤31）

8086 中有 8 条移位指令，移位计数使用 CL 或 1 表示，且规定当移位次数大于 1 时，必须使用 CL。从 80286 开始，则修改了上述的限制，当移位次数为 1～31 时，允许使用立即数。例如：

```
ROL   AX, 5
SHL   WORD   PTR [BX], 18
```

（2）SHLD　寄存器/存储器，寄存器，CL/立即数

将第一操作数（16 位或 32 位通用寄存器或存储单元）左移若干位（左移位数由 8 位立即数或 CL 指定），空出位用第二操作数（与第一操作数长度相同的通用寄存器）高位部分填补，但第二操作数的内容不变，CF 标志位中保留第一操作数最后的移出位。若仅移一位，当 CF 值与移位后的第一操作数的符号位不一致时，OF 置 1；否则 OF 置 0。移位过程如图 3.23 所示。

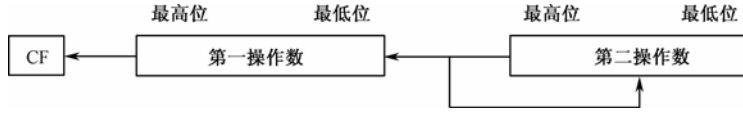


图 3.23　SHLD 指令功能

【例 3.35】
```
MOV   AX, 8321H
MOV   DX, 5678H
SHLD   AX, DX, 1        ; （AX）=0642H, DX=5678H, CF=1, OF=1;
SHLD   AX, DX, 2        ; （AX）=1909H, DX=5678H, CF=0, OF=0。
```

（3）SHRD　寄存器/存储器，寄存器，CL/立即数

将第一操作数（16 位或 32 位通用寄存器或存储器单元）右移若干位（右移位数由 CL 或 8 位立即数指定），空出位用第二操作数（与第一操作数长度相同的通用寄存器）低位部分填补，指令执行后，第二操作数内容不变，CF 标志位中保留第一操作数最后的移出位。移位过程如图 3.24 所示。

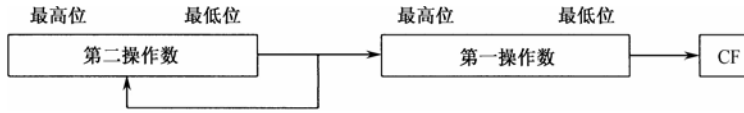


图 3.24　SHRD 指令功能

【例 3.36】
```
MOV   AX, 4B02H         ; AX=0100 1011 0000 0010
MOV   BX, 6040H         ; BX=0110 0000 0100 0000
SHRD   AX, BX, 7        ; AX 右移 7 位, BX 的低 7 位移入 AX 中, 结果为 AX=1000 0000
                          1001 0110（8096H）, BX=6040H（保持不变）, CF=0。
```

指令执行情况如图 3.25 所示。

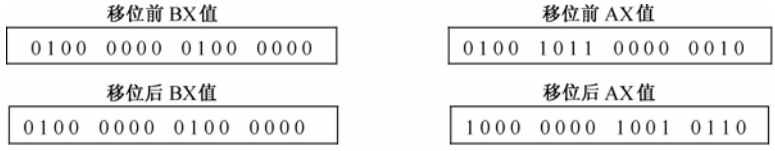


图 3.25　指令执行情况

6. 位操作指令

（1）位测试及设置指令

测试指令可用来对指定位进行测试，因而可根据该位的值来控制程序流的执行方向，而置位指令可对指定的位进行设置。

① BT　寄存器/存储器地址，寄存器/立即数

第一操作数（16 位或 32 位通用寄存器或存储单元）指定要测试的内容，第二操作数（与第一操作数同长度的通用寄存器或 8 位立即数）指定要测试的位，将被测内容的指定测试位的值送 CF，其他状态标志不确定。

【例 3.37】设 BX 指向一个存储单元数，CX 值为 4。

```
BT   [BX], CX          ; 检查由 BX 指向数据的第 4 位, 并将该位值送入 CF;
JC   Swhere            ; 若 CF=1, 则转移。
```

若用 8086 指令完成，可写成：

```
MOV   AX, [BX]         ; 将 BX 指向的数装入 AX 中;
TEST  AX, 10H          ; 检查第 4 位是否为 1;
JNZ   Swhere           ; 若 CF=1, 则转移。
```

② BTC　寄存器/存储器地址，寄存器/立即数

该指令在 BT 指令功能的基础上，将被测位取反。

③ BTR　寄存器/存储器，寄存器/立即数

该指令在 BT 指令功能的基础上，将被测位清 0。

④ BTS　寄存器/存储器,寄存器/立即数

该指令在 BT 指令功能基础上，将被测位置 1。

【例 3.38】

```
MOV   AX, 1234H
BT    AX, 2            ; （AX）=1234H, CF=1
BTC   AX, 2            ; （AX）=1230H, CF=1
BTR   AX, 2            ; （AX）=1230H, CF=0
BTS   AX, 2            ; （AX）=1234H, CF=0
```

（2）位扫描指令

位扫描指令用于找出寄存器或存储器地址中所存数据的第一个或最后一个是 1 的位。该指令可用于检查寄存器或存储器或存储单元是否为 0。

① BSF　寄存器，寄存器/存储器

对第二操作数（16 位或 32 位通用寄存器或存储器）从最低位到最高位进行扫描，将首先扫描到的“1”的位号送第一操作数（与第二操作数位数相同的通用寄存器），且使 ZF 置 0。若第二操作数的各位均为 0，则第一操作数的值不确定，且使 ZF 置 1。其他状态标志位

不确定。

② BSR　寄存器，寄存器/存储器

与 BSF 指令功能基本相同，唯一区别在于，该指令是从最高位到最低位进行扫描。

例如：
```
MOV   EAX, 01234567H
BSR   ECX, EAX          ; （ECX）=18H, ZF=0
BSF   AX, CX            ; （AX）=03H, ZF=0
```

7. 条件设置指令

这组 80386 特有的指令用于测试指定的标志位所处的状态，并根据测试结果，将指定的一个 8 位寄存器或内存单元置 1 或置 0。它们类似于条件转移指令中的标志位测试，但前者根据测试结果将操作数置 1 或置 0，而后者根据测试结果决定转移还是不转移。

指令格式为

SET 条件　寄存器/存储器

说明：条件是指令助记符的一部分，用于指定要测试的标志位。例如

```
SETZ    AL                  ; 当 ZF=1, 则（AL）=1, 否则（AL）=0
SETNC   BYTE  PTR [BX]      ; 当 CF=0, 则 BX 所指字节单元内容为 1
```

8. 内存范围检查指令

格式：BOUND　16 位寄存器，32 位存储器

以 32 位存储器低两字节的内容为下界，高两字节的内容为上界。若 16 位寄存器的内容在此上、下界表示的地址范围内，程序正常执行；否则产生 INT 5 中断（DOS 并未提供该类型中断处理程序，使用时用户需自行编写）。当出现这种中断时，返回地址指向 BOUND 指令，而不是 BOUND 后面的指令，这与返回地址指向程序中下一条指令的正常中断是有区别的。

【例 3.39】BOUND 指令的应用。

```
DATA   SEGMENT
BOTTOM      EQU   0
TOP         EQU   19
ANS         LABEL    DWORD          ; 给 ANS 分配 32 位地址;
WANS        DW    BOTTOM, TOP       ; 对边界初始化;
BOFF        DB     TOP +1   DUP（?）   ; 分配数组;
DATA    ENDS
CODE    SEGMENT
                M                   ; 假设 SI 为数组的地址指针;
        BOUND   SI, ANS             ; 检查 SI 是否在数据地址范围内, 否则进入类型 5 中断;
        MOV     DX , BOFF[SI]       ; 若在范围内, 继续使用。
                M
CODE    ENDS
```

3.5.2 Pentium新增指令

1. 字节交换指令

格式：BSWAP　寄存器

将 32 位通用寄存器以字节为单位进行高低字节的交换，即对指定寄存器的 32 位操作数的位 31～24 与位 7～0，位 23～16 与位 15～8 交换。该指令不影响状态标志位。

【例 3.40】寄存器（EDX）=1234 5678H

```
BSWAP    EDX
```

则指令执行后，（EDX）=7856 3412H。

80X86 系列处理器按“高高低低”的原则存储多字节数据，但某些处理器按“低低高高”的原则存储数据，BSWAP 指令特别适宜于这两种数据格式之间的转换。

2. 互换并相加指令

格式：XADD　寄存器/存储器，寄存器

该指令将第一操作数（8 位，16 位或 32 位通用寄存器或存储单元）与第二操作数（与第一操作数位数相同的通用寄存器）内容互换，并将两者之和送第一操作数。该指令对状态标志位的影响与 ADD 指令相同。

【例 3.41】若 BX 所指单元内容为 1122 3344H,（EAX）=0022 4466H,

```
XADD   [BX], EAX
```

指令执行后，（EAX）=1122 3344H，而 BX 所指单元内容为 1144 77AAH。XADD 指令功能相当于 XCHG 和 ADD 这两条指令的功能。该指令允许使用 LOCK 前缀。

3. 比较并交换指令

（1）格式：CMPXCHG　寄存器/存储器，寄存器

该指令将第一操作数（8 位，16 位或 32 位通用寄存器或存储单元）内容与对应长度的累加器（AL，AX 或 EAX）内容作比较，若相等，则使 ZF 置 1，且将第二操作数（与第一操作数位数相同的通用寄存器）内容送第一操作数；否则使 ZF 清 0，且第一操作数送对应累加器。

【例 3.42】CMPXCHG　ESI，EBX

若（ESI）=（EAX），则 ZF=1，且将（EBX）送 ESI；否则 ZF=0，且将（ESI）送 EAX。

（2）8 字节比较交换指令

格式：CMPXCHG8B　存储器

将 EDX：EAX 中的 8 字节值与指定的 8 字节存储器操作数相比较，若相等，则使 ZF 置 1，且将 ECX：EBX 中的值送指定的 8 字节存储单元替换原有存储器操作数；否则使 ZF=0，且将指定的 8 字节存储操作数器送 EDX：EAX。

【例 3.43】设 BX 所指 8 字节存储单元内容为 0011 2233 4455 6677H，（EDX）=0，（EAX）=FFFF FFFFH。

```
CMPXCHG8B   [BX]
```

则该指令执行后，（EDX）=0011 2233H，（EAX）=4455 6677H，ZF=0。

4. Cache管理指令

（1）使整个片内 Cache 无效指令

格式：INVD

该指令用于将 CPU 内部 Cache 的内容无效。其具体的操作是刷新内部 Cache，并分配一个专用总线周期刷新外部 Cache，执行该指令不会将外部 Cache 中的数据写回主存，即 Cache 中数据自然丢失。

（2）写回并使 Cache 无效指令

格式：WBINVD

该指令功能与 INVD 相似，具体操作是刷新内部 Cache。并分配一个专用总线周期将外部 Cache 的数据写回主存，并在此后的一个专用总线周期将外部 Cache 刷新。

（3）使 TLB 无效指令

格式：INVLPG

该指令使页式管理机构内的高速缓冲器 TLB 中的某一项作废。若 TBL 中含有一个存储器操作数映象的有效项，则该 TBL 项被标记为无效。

5. 处理器特征识别指令

格式： CPUID

根据 EAX 中的参数，将处理器的说明信息送 EAX，特征标志字送 EDX。

6. 读时间标记计数器指令

格式： RDTSC

将 Pentium 中的 64 位时间标记计数器的高 32 位送 EDX，低 32 位送 EAX。该计数器随每一个时钟递增，在 Reset 后该计数器被置 0。利用该计数器可检测程序运行性能。

7. 读模型专用寄存器指令

格式： RDMSR

将 ECX 所指定的模型专用寄存器的内容送 EDX、EAX，具体来说，高 32 位送 EDX，低 32 位送 EAX。若所指定的模型寄存器不是 64 位，则 EDX、EAX 中的对应位无定义。

8. 写模型专用寄存器指令

格式： WRMSP

将 EDX、EAX 的内容送到由 ECX 指定的模型专用寄存器。具体来说，EDX 和 EAX 的内容分别作为高 32 位和低 32 位。若指定的模型寄存器有未定义或保留的位，则这些位的内容不变。

本小节只是简单介绍了 Pentium 新增指令，对这几条指令的理解和应用要求具备 Pentium 体系结构方面的知识，请参阅有关文献资料。

习　题

3.1　分别指出下列指令中源操作数和目的操作数的寻址方式。

1）MOV　AX，0100H　　2）MOV　[SI]，AL

3）ADC　BL，[2000H]　　4）AND　BYTE　PTR [2000H]，1

5）MOV　2[BX][DI]，DX　　6）OR　AX，80H [SI]

3.2　指出以下指令是否合法。

1）MOV　CS，AX　　2）MOV　[2000H]，AL

3）ADD　[2000H]，40H　　4）MUL　AX，BX

5）AND　184CH，AX　　6）MOV　DS，0

7）INC　[BX]　　8）SBB　AX，[DX]

9）RCL　BX，2　　10）POP　AL

11）XCHG　BX，4050H　　12）CWB

3.3　试根据以下要求写出相应的汇编语言指令。

1）将 CX 寄存器的内容与 DX 寄存器的内容相加，结果存入 DX 寄存器中。

2）用寄存器 BX 和 DI 的基址变址寻址方式把存储器中的一个字节数据与 AH 寄存器的内容相加，并把结果存入 AH 寄存器中。

3）用寄存器 BX 和位移量 2000H 的寄存器相对寻址方式把存储器的一个字数据和（DX）相加，并把结果送回存储器中。

4）将数 0C3H 与（BL）寄存器相加，结果送回 BL 寄存器中。

5）将存储器数据段中 1300H 单元（采用直接寻址方式）中的一个字数据与立即数 3456H 相减，结果送回存储器 1300H 单元。

3.4　假设 AX 中的数据为 6987H，DATE1 分别为下列数值时，执行 ADD　AX，DATE1 指令后，标志位 SF、ZF、CF 和 OF 的状态是什么？

1）1234H　　2）4801H

3）EB30H　　4）902AH

3.5　假设 AX 中的数据为 3760H，DATE1 分别为下列数值时，执行 CMP　AX，DATE1 指令后，标志位 SF、ZF、CF 和 OF 的状态是什么？

1）1234H　　2）4801H

3）EB30H　　4）902AH

3.6　设（SP）=2040H，（AX）=12D4H，（BX）=36F4H。试回答：

1）执行 PUSH　AX 指令后，（SP）=?

2）再执行 PUSH　BX 及 POP　AX 指令后，（SP）=？，（AX）=?

3.7　写出执行以下计算的指令序列，其中 X，Y，Z，R 和 Q 均为存放 16 位带符号数单元的地址。

1）Q←X+（Z–Y）

2）Q←X+（Y+9）–（Z+3）

3）Q←(X*Y)+（Z/R）

4）Q←5*（X–Y）+R/（Z+8）

3.8　现有（DS）=4000H，（BX）=0100H，（SI）=0002H，（40100H）=11H，（40101H）=22H，（40102H）=33H，（40103H）=44H，（41200H）=55H，（41201H）=66H，（41202H）=77H，（41203H）=88H，试说明下列各条指令执行完成后 AX 的内容。

1）MOV　AX，0100H

2）MOV　AX，[BX]

3）MOV　AX, 1200H

4）MOV　AX, BX

5）MOV　AX, [BX][SI]

6）MOV　AX, 1100H[BX]

7）MOV　AX, 1100H[BX][SI]

8）MOV　AX, 2[BX]

3.9　已知：(DS)=2000H，(ES) =3000H，(CS)=5000H，（SS）=4000H (AX)=5566H，(BX)=1000H，(BP)=0010H，(21000H)=1122H，(31000H)=7766H，(40010H)= 0FFFFH。

请写出下列各条指令独立执行完后，有关寄存器及存储单元的内容。

1）ADD　AL , [BX]

2）AND　AX , BX

3）SUB　ES:[BX],AX

4）XOR　[BP] , AX

3.10　在数据段 DATA 中已为 0120H 单元定义的符号名为 MESS，其中存放的数据为 1122H。

1）试问以下两条指令有什么区别，指令执行结束后 AX 寄存器的内容是什么。

```
MOV　AX, MESS
LEA　AX, MESS
```

2）试问以下两条指令有什么区别，指令执行结束后 AX 寄存器的内容是什么。

```
MOV　AX, MESS
MOV　AX, OFFSET　MESS
```

3.11　写出各种使 AL 置 0 的指令。

3.12　用两种方法实现将（AL）乘以 10 值送 AX 的功能。

3.13　写一程序段，将附加段 2000H～2003H 四个字节之和送 AX。

3.14　假设（DX）=0F7H，变量 DATA1 中内容为 9EH，确定下列每条指令执行后的结果。

1）AND　DX, DATA1

2）XOR　DX, DATA1

3）OR　DX, DATA1

4）SHL　DX, 1

5）XOR　DX, 0FFH

6）AND　DX, 0H

7）TEST　DX, 80H

8）TEST　DX, 01H

3.15　试写出移位指令执行后 BX 寄存器的内容，执行前（BX）=6CB5H，CF=0。

1）MOV　CL, 04H

```
SHR　BX, CL
```

2）MOV　CL, 03H

```
SAL　BX, CL
```

3）ROR　BX, 1

4）MOV　CL, 06H

RCR　BX, CL

3.16　若（AX）=0012H，（BX）=0034H，则下列指令执行后 AX 为多少？

```
MOV　CL, 8
ROL　AX, CL
ADD　AX, BX
```

3.17　试分析下面的程序段所完成的功能。

```
MOV　　CL, 4
SHL　　DX, CL
MOV　　BL, AH
SHL　　AX, CL
SHR　　BL, CL
OR　　DL, BL
```

3.18　下列程序段执行后（AL）=？（DL）=？完成的是什么功能？

```
MOV　CL, 4
MOV　AL, 48H
MOV　DL, AL
AND　AL, 0FH
OR　AL, 30H
SHR　DL, CL
OR　DL, 30H
```

3.19　编写程序使：

1）BX 寄存器低 4 位置 1

2）AX 寄存器的低 4 位清 0

3）AX 寄存器各位取反

4）CX 寄存器的低 4 位取反

5）用 TEST 指令测试 AL 寄存器的位 1、位 5 和位 7 是否同时为 1，如果是则把 0FFH 送 CL 寄存器，否则将 0 送 CL 寄存器。

6）将 AH 的低 4 位与 AL 的低 4 位拼成一个字节（AH 的低 4 位为拼装后的高 4 位），结果送 AH 中。

3.20　说明 80386 的 32 位通用寄存器与 16 位通用寄存器这间的关系。

3.21　80386 的寻址方式有何特点。

3.22　设 AX 中有一有符号数，请用两条不同的指令将 AX 中的值扩展到 EAX 中。

3.23　请用两条不同的指令使 BX 所指的 32 位存储器操作数的第 20 取反。

3.24　用一条指令实现把 EAX 中的 32 位数保存到寄存器对 DX：AX 中。

3.25　设寄存器对 EDX：EAX 中放有 0102030405060708H，请写出指令使其中的内容成为 0807060504030201H。

3.26　试给出下列指令序列执行后目的寄存器的内容。

1）MOV　BX, –10H

MOVSX　EBX, BX

2）MOV　DL, –37H

MOVSX　ECX, DL

3）MOV　AX, 37H

MOVZX　ECX, AX

4）MOV　CL, 0A3H

MOVZX　EAX, CL

3.27　编写程序段，将 EBX、ECX 和 ESI 寄存器的内容相加，其和存入 EDI 寄存器中（不考虑溢出）。

第 4 章　汇编语言与汇编语言程序

本章主要介绍汇编语言程序的格式和组成元素，对于表达式和常用伪指令进行了较为详细的讲述，还介绍了汇编语言程序的上机过程。

4.1　汇编语言程序与汇编程序

用汇编语言编写的程序称为汇编语言源程序，简称汇编语言程序。

汇编语言程序虽然比机器语言程序直观、易懂、便于交流和维护，但不能像机器语言那样为计算机直接识别和执行，这一点与许多高级语言源程序相似。汇编语言源程序只有被编译成机器语言程序后，才能被计算机执行。这种机器语言程序称为目标程序（Object Program）。虽然目标程序已经是二进制文件了，但是它还不能直接上机执行，必须经过连接程序把目标文件与库文件或其他目标文件连接在一起形成可执行文件，才可以由 DOS 装入存储器，并在机器上执行。

将汇编语言源程序编译成目标程序的加工程序称为汇编程序。这一加工过程称为汇编，三者的关系如图 4.1 所示。

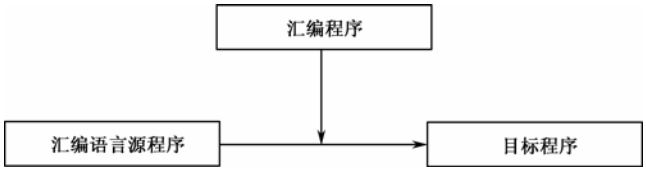


图 4.1　汇编语言源程序、目标程序及汇编程序之间的关系

说明：汇编程序与汇编语言程序是两个不同的概念。汇编程序是用来将汇编语言程序编译成机器代码的工具；汇编语言程序是用户根据实际需要编写的程序。

8086/8088、80286、80386、80486 及 Pentium 等系统广泛使用的汇编程序为 80X86 宏汇编程序，简称为 MASM（Macro-Assembler）。

4.2　汇编语言程序的格式和组成元素

为了说明汇编程序格式，这里给出了一个汇编语言示例程序。

【例 4.1】示例程序。

```
NAME    EXAMPLE
DSEG    SEGMENT
DATA1   DB   4   DUP（1）, 10H, 11, 0AH, 0, 0BH
SUM     DB   ?
COUNT   EQU  9
DSEG    ENDS
```

```
SSEG    SEGMENT  STACK
        DB   100H   DUP（?）
SSEG    ENDS
CSEG    SEGMENT
        ASSUME   CS: CSEG, DS: DSEG, SS: SSEG
START:  MOV   AX, DSEG
        MOV   DS, AX
        XOR   AL, AL            ; AL 清 0;
        MOV   CX, COUNT         ; 设置相加次数, 也即循环操作次数;
        LEA    SI, DATA1        ; SI 指向数据区起始位置;
LOOP1:  ADD   AL, [SI]          ; 将 SI 所指数据加到 AL 中;
        INC    SI               ; SI 指向下一字节;
        LOOP   LOOP1            ; CX 减 1 计数, 减 1 后不为 0 则转至 LOOP1 标号处;
        MOV   SUM, AL           ; 将 DATA1 中 9 个字节之和送变量 SUM;
        MOV   AH, 4CH
        INT   21H               ; 返回 DOS。
CSEG    ENDS
        END   START
```

这一程序实现了 DATA1 数据区中 9 个字节的相加，基本上体现了汇编语言程序的一般格式和组成元素。

（1）汇编语言程序采用以 SEGMENT 和 ENDS 定义的段结构，一个汇编语言程序由若干个段组成。

（2）段中包含两种语句：指令语句，即前面介绍的指令，以及伪指令语句，简称伪指令。

（3）语句中包含若干元素：标识符，保留字，表达式等。

以下介绍汇编语言程序的组成元素，指令语句已在前面讲解，此处不再重复。

4.2.1 标识符

标识符是源程序中便于指定和阅读的字符串。例如，示例程序中的数据段名 DSEG，变量 DATA1，标号 LOOP1，符号常量 COUNT 等都是标识符。

标识符可以由字母 A～Z，a～z，数字 0～9，专用字符？、.、@、$、_（下画线）等字符组成。除数字外，所有这些字符均可作为标识符的首字符。“.”只能作为标识符的首字符。标识符可由多个字符组成，但仅前 31 个字符能为汇编程序识别。

4.2.2 保留字

保留字是汇编程序预留的具有固定用途的字符串。例如，示例程序中的 SEGMENT，DB，MOV，CX 等所有寄存器名，指令及伪指令助记符，运算符等均属于保留字。在编写源程序时，一般要避免将保留字用于非固定用途。

4.2.3 表达式

表达式是由常量、变量、标号及运算符等构成的式子。表达式分为数值表达式和地址表

达式。

1. 常量

常量分为字面常量、符号常量及串常量。字面常量由 0～9，A，B，C，D，E，F 以及基数后缀，即尾标 B，D，H，Q（或 O）构成。如果一个十六进制常量以字母开头，则须在前面添加数字 0 以区别于标识符。例如，示例程序中的 10H，0A1H，9 等均属于字面常量，如果将 0A1H 中的数字 0 去掉，即写成 A1H，则会被当做以字母 A 开头的标识符。

符号常量是使用 EQU、= 伪指令定义其值的标识符。例如，示例程序中的标识符 COUNT 被定义为数值 9 。

串常量是用单引号括起的一串字符。串常量以单引号中各字符的 ASCII 码存储。例如，'Data' 以 44H，61H，74H，61H 存储。

2. 变量

在汇编语言程序中的变量是存储单元的标识符，即数据存放地址的符号表示。例如，示例程序中的 DATA1 就是变量。对变量的访问是通过变量名来实现的。因此，变量名被认为是变量的符号地址。变量名一般由定义变量的伪指令语句确定。

由于存储器是分段使用的，所以源程序定义的变量具有 3 个方面的属性。

（1）变量的段属性

变量的段属性是指变量所在段的段地址。当需要访问该变量时，该段地址一定要在某一段寄存器中。例如，在示例程序中，通过指令：

```
MOV   AX, DSEG
MOV   DS, AX
```

将变量 DATA1，SUM 所在段的段地址放到 DS 中，以便其后对这些变量进行访问。

（2）变量的偏移属性

变量的偏移属性是指变量所在段的首地址到该变量的偏移量。例如：示例程序中变量 DATA1 和 SUM 的偏移地址分别为 0000H 和 0009H。

（3）变量的类型属性

变量的类型是指存取该变量中的数据所对应的字节数。有字节（BYTE）、字（WORD）、双字（DWORD）、四字（QWORD）及十字节（TBYTE）等类型。变量类型由变量定义时所使用的伪指令决定。例如，示例程序中通过 DB 伪指令将变量 DATA1、SUM 定义为字节类型。

3. 标号

汇编语言程序中的标号是机器指令存放位置的标识符，即机器指令存放地址的符号表示。它可作为转移指令或重复控制指令转向目的操作数。例如，示例程序中的 LOOP1 就是标号，它作为 LOOP 指令的目的操作数。标号一般只在代码段中定义和引用。由于标号代表了指令的符号地址，所以标号也有 3 个属性：

（1）标号的段属性

标号的段属性是指标号定义所在段的段地址。

（2）标号的偏移属性

标号的偏移属性是指标号所在段的首地址到该标号的定义语句的偏移量。

（3）标号的类型属性

标号的类型有两种：NEAR 类型和 FAR 类型。NEAR 类型的标号只能在定义该标号的段内使用，而 FAR 类型的标号则无此限制。

4. 数值表达式

数值表达式是由常量与算术运算符、逻辑运算符或关系运算符构成的式子。下面分别介绍这 3 类运算符，运算符见表 4.1 所示。

表 4.1　数值表达式中常用的运算符

算术运算符	逻辑运算符	关系运算符
+ （加法）	NOT （非）	EQ （相等）
- （减法）	AND （与）	NE （不相等）
* （乘法）	OR （或）	LT （小于）
/ （除法）	XOR （异或）	GT （大于）
MOD （求余）	SHL （左移）	LE （小于或等于）
	SHR （右移）	GE （大于或等于）

（1）算术运算符

算术运算符包括+、一、*、 / 、MOD（模除）。其中， / 运算的结果为两常量之商的整数部分。如：122/6 的结果为 20。MOD 运算的结果为两常量相除所得余数。如：122 MOD 6 的结果为 2。

（2）逻辑运算符

逻辑运算包括 NOT，AND，OR， XOR，SHL 及 SHR。这些运算符实现相关常量的二进制逻辑运算。例如：

NOT　12H 的结果为 0EDH;

1234H　AND　0FH 的结果为 04H;

1234H　 OR　0FH 的结果为 123FH;

1234H　XOR　1FH 的结果为 2BH。

SHL 或 SHR 运算是指将运算符左边的二进制常量左移或右移运算符右边所指定的位数（正整数），空出位补 0。如：0010B　SHL 2 的结果为 1000B。

（3）关系运算符

关系运算符包括 EQ（相等）、NE（不等）、LT（小于）、GT（大于）、LE（小于等于）、GE（大于等于）。此类运算的结果是两个特殊的常量，若关系成立则结果为–1（用补码表示），否则为 0。例如

设已定义符号常量 COUNT 的值为 9，则 COUNT NE 2 的值为–1，COUNT GE 10 的值为 0。

数值表达式的运算在汇编期间进行，且运算结果为数值常量。

5. 地址表达式

地址表达式是由常量、变量、标号、寄存器 BP、BX、SI、DI 内容（在此用方括号括起）和运算符组成的式子。例如，本书第三章中所述的直接寻址、寄存器间接寻址、基址寻址、变址寻址、基址变址寻址的表示均属地址表达式。又如，示例程序中的［SI］也属于地址表

达式。地址表达式的值一般为段内偏移地址，具有段属性、偏移属性及类型属性。

地址表达式除使用前述运算符外，还可以使用表 4.2 所示的运算符。

（1）属性定义算符

① 段超越前缀“：”。该运算符用于给变量、标号或地址表达式临时指定一个段属性，其格式为

段寄存器名：地址表达式

或段名：地址表达式

例如：

```
MOV   AL, ES:［1000H］   ; 以（ES）作为段地址, 以 1000H 作为偏址的存储单元的内容送 AL。
MOV   AL, [1000H]        ; 以（DS）作为段地址, 以 1000H 作为偏移地址的存储单元的内容送
                           AL。如前所述, ［1000H］的段属性默认为（DS）。另外, 并不因前
                           一指令中使用过段超越前缀符而改变本语句中[1000H]的段属性。
```

表 4.2　地址表达式中专用运算符

属性运算符	分析运算符	分离运算符	其他运算符
：（段超越前缀）	SEG（取段地址）	HIGH	SHORT
PTR（类型运算）	OFFSET（取偏移地址）	LOW	（）
THIS（定义类型）	TYPE（取类型）		[ ]
	LENGTH（取变量单元数）		
	SIZE（取变量总字节数）		

② 类型运算符 PTR。该运算符用于给变量、标号或地址表达式临时指定一个类型。其格式为

类型　PTR　地址表达式

根据地址表达式的不同值，类型可以是 BYTE，WORD，DWORD，NEAR 及 FAR 等。例如：

```
MOV   WORD   PTR[2000H], 0   ; 将偏移地址为 2000H 的字单元, 即 2000H 和 2001H 两字节清 0。
MOV   BYTE   PTR[2000], 0    ; 将偏移地址为 2000H 的字节清 0。
```

③ 定义类型运算符 THIS。该运算符具有与 PTR 类似的功能，可用于指定某个变量、标号或地址表达式的类型，但在具体用法上有区别。其格式为

标识符　EQU　THIS　类型

将 THIS 右边的类型赋给左边的标识符。

（2）分析运算符

① 取段地址运算符 SEG。该运算符产生其后的变量或标号所在段的段地址。例如

```
MOV   AX, SEG DATA1          ; 将变量 DATA1 所在段的段地址送 AX。
```

② 取偏移地址运算符 OFFSET。该运算符产生其后的变量或标号的偏移地址。例如

```
MOV   BX, OFFSET   DATA1     ; 将变量 DATA1 的偏移地址送 BX。
```

③ 取类型运算符 TYPE。该运算符产生其后的变量或标号的类型值。变量或标号的类型与该运算符产生的类型值的对应关系如表 4.3 所示。

④ 取变量单元数运算 LENGTH。该运算符产生其后变量所包含的单元个数。该运算的结果根据该变量定义伪指令中第一个表达式的形式而定。若第一个表达式为重复子句“n DUP

（数值表达式）”，则 LENGTH 运算的结果为重复因子 n；否则结果为 1 。伪指令中“n　DUP（数值表达式）”等同于将括号中的表达式重复 n 次。例如，示例程序中定义变量 DATA1 的伪指令

```
DATA1　DB　4　DUP（1）, 10H, 11H, 0AH, 0, 0BH
```

中第一个表达式为“4　DUP（1）”，所以 LENGTH　DATA1 的值为 4。

表 4.3　存储器操作数的类型值

变量类型	类型值	标号类型	类型值
BYTE	1	NEAR	–1
WORD	2	FAR	–2
DWORD	4		
QWORD	8		
TBYTE	10		

⑤ 取变量总字节数运算符 SIZE　该运算符产生其后变量所包含的总字节数。例如：对于示例程序中的变量 DATA1 而言，SIZE　DATA1 的值为 9。

又如，设有伪指令

```
DATA2　DW　4　DUP（1）, 10H, 11H, 0A1H, 0, 0BH
```

也即将 DATA2 定义为字变量，则 4 个 1 以及 10H，11H，0A1H，0 及 0BH 各占两个字节,那么 SIZE DATA2 的值为 18。

【例 4.2】分析运算符应用举例。

下面定义的数据段 DATA，假设段地址为 2000H，则利用分析运算符可进行如下运算。

```
DATA　SEGMENT
D1　　DB　41H, 6DH
D2　　DW　803AH, 104FH
D3　　DD　12345678H, 0ABCDEF9H
D4　　DW　40　DUP（1）
DATA　ENDS
```

运算一：

```
MOV　AX, SEG　D1　　; AX=2000H
MOV　BX, SEG　D2　　; BX=2000H
MOV　DX, SEG　D3　　; DX=2000H
```

变量 D1，D2 和 D3 同属一个段，故它们的段地址相同。

运算二：

```
MOV　AX, OFFSET　D1　　; AX=0
MOV　BX, OFFSET　D2　; BX=2
MOV　DX, OFFSET　D3　　; DX=6
```

变量 D1，D2 和 D3 的偏移地址分别是 0、2 和 6。

运算三：

```
MOV　AX, TYPE　D1　　; AX=1
MOV　BX, TYPE　D2　　; BX=2
```

```
MOV   DX, TYPE   D3          ; DX=4
```

变量 D1，D2 和 D3 的类型值分别是 1、2 和 4。

运算四：

```
MOV   AX, LENGTH   D4       ; AX=40
MOV   BX, SIZE   D4         ; BX=80
```

而：

```
MOV   AH, LENGTH   D1       ; AH=1
MOV   AL, SIZE   D1         ; AL=2
MOV   BH, LENGTH   D2       ; BH=1
MOV   BL, SIZE   D2         ; BL=4
MOV   DH, LENGTH   D3       ; DH=1
MOV   DL, SIZE   D3         ; DL=8
```

（3）分离运算符

① 分离高字节运算符 HIGH。该运算符产生其后的运算对象的高字节。例如

```
MOV   AL, HIGH   1234H      ; 将 12H 送 AL。
```

② 分离低字节运算符 LOW。该运算符产生其后的运算对象的低字节。例如

```
MOV   AL, LOW   1234H       ; 将 34H 送 AL。
```

（4）其他运算符

SHORT　运算符用于说明其后的标号在短矩离（–128～127 之间）内；

（）运算符用于改变运算的优先级别；

[ ]　运算符用于表示间接寻址等。

各种常用运算符的优先级别见表 4.4 所示。

表 4.4　运算符的优先级

运　算　符	优 先 级 别
LENGTH、SIZE、括号（圆括号、方括号、尖括号）	高
:（段超越前缀符）、PTR、THIS、SEG、OFFSET、TYPE	↓
HIGH、LOW	
*、/、MOV、SHL、SHR	
+、–	
EQ、NE、LT、LE、GT、GE	
NOT	
AND	
OR、XOR	
SHORT	低

4.3　伪指令

汇编语言程序由多条语句构成。而语句分为指令语句和伪指令语句，分别简称为指令和伪指令。在第三章和第四章中已分别介绍了 8086/8088、80286、80386、80486、及 Pentium 中的指令。指令在汇编过程中被翻译成相应的目标代码，并经过连接后生成计算机可执行的机器指令代码。伪指令属于汇编控制命令，在汇编过程中实现数据定义、分配存储区、指示

程序结果等功能。伪指令本身一般不产生任何目标代码。

80x86 宏汇编语言提供了多种伪指令，本节只介绍部分常用伪指令。

4.3.1 符号定义伪指令

符号定义伪指令用于为程序中多次出现的同一个常量或表达式定义一个标识符，以便在源程序中以标识符来代替对应的常量或表达式。

1. EQU 伪指令

格式：标识符 EQU 表达式

功能：用表达式来定义标识符，即使得标识符等同于表达式。

例如：
```
COUNT   EQU   9                  ; 定义一个符号常量 COUNT, 使之等同于 9;
ADDR    EQU   ES：[BX][SI]    ; 使 ADDR 等同于地址表达式 ES：[BX][SI]。
```

2. = 伪指令

格式：标识符 = 表达式

功能：与 EQU 伪指令功能基本相同，但两者中只有 = 伪指令可对同一标识符作重新定义。

例如：
```
COUNT = 9                ; 定义 COUNT 等于 9;
COUNT = COUNT+1      ; 重新定义 COUNT 等于 10。
```
再如：
```
DT1    EQU    30
DT1    EQU    20        ; 此定义错误, 因为前面已经用 EQU 伪指令对 DT1 作了定义。
```
注意：EQU、 = 伪指令仅仅是对程序中某些符号进行等价说明，并不实际分配存储单元，因此，用 EQU、 = 伪指令定义的符号不占存储单元。

4.3.2 变量定义伪指令

变量定义就是为数据分配存储单元，有时还为这个存储单元取一个变量名。

1. DB伪指令

格式：[变量] DB 一个或多个表达式。

功能：告知汇编程序，留出一块内存单元作为字节数据区，并在其中存放各表达式的值，先出现者对应低地址，后出现者对应高地址。若此伪指令中设有 DB 左边的“变量”，则用此变量来标识新定义的内存单元。表达式可以是以下 4 种形式。

（1）字节常量以及不确定常量“?”。

（2）重复子句：数值表达式 DUP（一个或多个表达式）。

（3）串常量。

（4）以上 3 种形式的任意组合。

例如：
```
DATA1    DB   0, 45H , 0FFH, ?
DATA2    DB   'HELLO!'
DATA3    DB   10 , 2 DUP （2 DUP （1, 2）, 3）
```
这里定义的字节变量在存储器中的存储格式如图 4.2 所示。

图 4.2 中“--”表示不确定常量，各个字符实际上以其 ASCII 码形式存储。

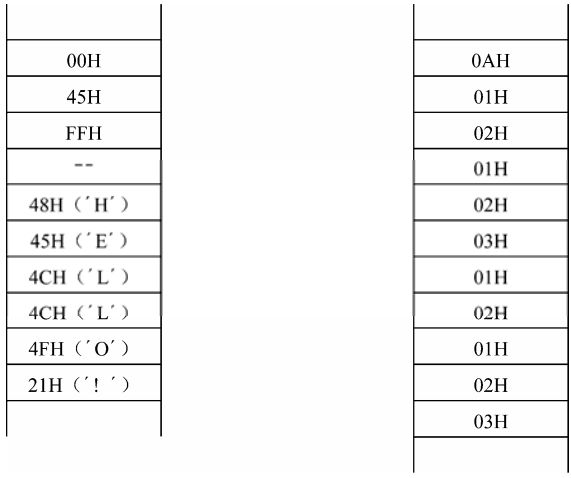


图 4.2　字节变量存储格式

2. DW伪指令

格式：[变量]　DW　一个或多个表达式。

功能：与 DB 伪指令功能类似，但定义的是字变量。表达式可以是以下 4 种形式。

（1）除了是字数据外，与 DB 伪指令的（1）、（2）同；

（2）地址表达式。在此情况下，实际取其偏移地址；

（3）一个或两个字符组成的串常量；

（4）以上 3 种形式的任意组合。

例如：
```
ARRAYW    DW    –1, 1234H, 2 DUP（–32768）
ADDR      DW    ARRAYW+2
STRING    DW    'EH', 'LL', '!O'
```

这里定义的字变量在存储器中的存储格式如图 4.3 所示（此处设第一个字变量的存储地址为 01A0：3000，即段地址为 01A0H，偏移地址为 3000H），当 DW 后有两个字符组成的串常量时，这两个字符中的后一个字符对应低地址，前一个字符对应高地址。

3. DD伪指令

格式：[变量]　DD　一个或多个表达式。

功能：与 DB 伪指令功能类似，但定义的是双字变量。表达式可以是以下 4 种形式。

（1）除了是双字数据外，与 DB 伪指令的（1）、（2）同。

（2）地址表达式。在此情况下，分别将偏移地址和段地址存放到存储器中，偏移地址对应较低地址。

（3）1～4 个字符组成的串常量。

（4）以上 3 种形式的任意组合。

除上述 3 种变量定义伪指令外，还有 3 字变量定义伪指令（DF 伪指令）、4 字变量定义伪指令（DQ 伪指令）和 10 字节变量定义伪指令（DT 伪指令）。

4.3.3　段定义伪指令

段定义伪指令指示汇编程序如何按段组织程序和使用存储器。

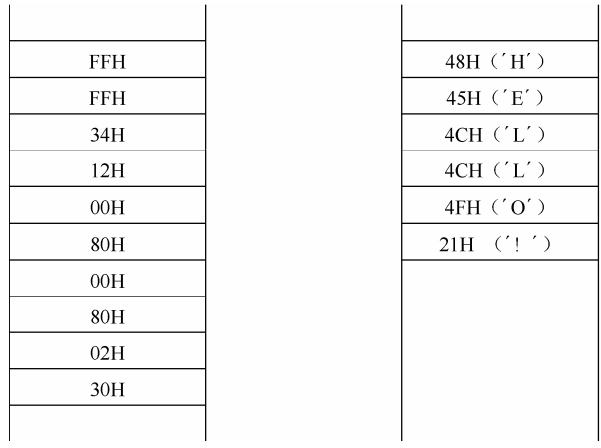


图 4.3　字变量存储格式

1. SEGMENT和ENDS伪指令

格式：段名 SEGMENT [定位方式][组合方式]['类别']

M　　；段体

段名 ENDS

功能：SEGMENT 和 ENDS 伪指令用于把程序分成若干逻辑段。这些逻辑段根据其用途的不同分为代码段、数据段、堆栈段和附加段，它们被分别装入由 CS、DS、SS 和 ES 所指定的物理段中。

段名由程序员指定，且起始处和结束处的段名一致。段体为段内的指令和伪指令序列。其后的 3 个参数一般可任选，它们的含义将在第十一章中介绍。

2. ASSUME 伪指令

格式：ASSUME　段寄存器：段名 [，段寄存器：段名…]

功能：该伪指令用于通知汇编程序，CS、DS、SS 或 ES 被设定为哪些段的段地址寄存器，从而在汇编时能知道语句中引用的变量、标号或表达式所对应的段。例如：示例程序中的 ASSUME 伪指令将 DS 设定为 DSEG 段的段地址寄存器，所以在对指令

LOOP1：ADD　AL, [SI]

进行汇编时，知道源操作数对应的段为 DSEG。

值得注意的是：ASSUME 伪指令只是告知汇编程序有关段寄存器将被设定为哪个段的段地址，而段寄存器必须通过指令来设定具体值。例如：示例程序中在使用上述 ASSUME 伪指令后，通过指令

MOV AX, DSEG

MOV DS, AX

给 DS 设定具体值。

3. ORG 伪指令

格式：ORG　表达式

功能：告知汇编程序，使其后的指令或数据从表达式的值所指定的偏移地址开始存放。表达式的值应为 0～65535（即 0000H～FFFFH）例如：ORG $+10 表示其后的指令或数据跳过 10 个字节存放。其中$表示当前偏移地址。

4.3.4 过程定义伪指令

在程序设计中，常把具有某种功能的程序段设计成一个过程。80x86 宏汇编语言用于过程定义的伪指令的格式为

```
过程名  PROC  [NEAR 或 FAR]
        M          ; 过程体
过程名  ENDP
```

其中过程名由程序员指定，且起始处和结束处的过程名一致。过程体为过程内的指令和伪指令序列。定义一个近过程时，参数 NEAR 可省略。

4.3.5 80x86 指令集选择伪指令

这类指令一般放在程序的起始位置，用来通知汇编程序以下程序中使用哪一种指令集，如表 4.5 所示。

表 4.5 汇编程序指示符

伪 指 令	功 能
• 8086	选择 8086/8088 指令集
• 286	选择 80286 指令集
• 286P	选择 80286 保护模式指令集
• 386	选择 80386 指令集
• 386P	选择 80386 保护模式指令集
• 486	选择 80486 指令集
• 486P	选择 80486 保护模式指令集
• 586	选择 Pentium 指令集*
• 586P	选择 Pentium 保护模式指令集*

注意：*用于 6.11 版本的 MASM。

说明：（1）8086 可默认。当一个程序未使用 80x86 指令集选择伪指令时，被默认为使用 8086/8088 指令集。（2）80x86 指令集向上兼容。例如，在起始位置使用了 • 386 伪指令的程序中可使用 8086/8088，80286 及 80386 指令集中的指令，但不能使用 80486 指令集中的指令。

4.4 汇编语言程序的上机过程

汇编语言程序的处理过程包括编辑、汇编、连接及执行。一般可以分为以下 4 个步骤：

（1）用编辑程序，建立扩展名为.ASM 的汇编语言源程序文件。

（2）用汇编程序将 ASM 文件汇编成二进制的目标文件，即 OBJ 文件。

（3）用连接程序，可将 OBJ 文件连接为可执行文件，即 EXE 文件。

（4）可在 DOS 环境下直接执行 EXE 文件，亦可通过 DEBUG 调试和执行。

该处理过程为图 4.4 所示。

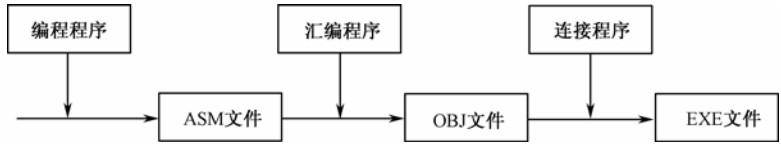


图 4.4　汇编语言程序的处理过程

本节将介绍上机实现上述过程的具体操作方法。

上机操作首先使用 EDIT 或其他文本编辑工具编辑源程序，建立扩展名为.ASM 的源程序文件。源程序一般一行安排一条语句。一般而言各语句的标号、助记符、操作数及注释首字符对齐，如【例 4.1】中示例程序所示。在【例 4.1】示例程序被编辑后，建立 EXAMPLE.ASM 源程序文件。

4.4.1　MSAM汇编环境

1. 生成OBJ文件

可使用常用 Microsoft 公司的 MASM 汇编程序根据已建立的 ASM 文件生成对应的 OBJ 文件。汇编语言源程序一定要用.ASM 作为扩展名，否则 MASM.EXE 文件不进行汇编。现以 EXAMPLE.ASM 文件汇编为 EXAMPLE.OBJ 文件为例说明生成 OBJ 文件的过程。具体操作步骤如下。

（1）通过鼠标导航到 MASM.EXE 文件所在的文件夹下；

（2）在 MASM.EXE 文件的位置上双击左键，启动汇编程序；

（3）在汇编时，用户根据提示输入源程序文件名 EXAMPLE.ASM；

（4）根据提示输入其他有关信息。

其操作过程如图 4.5 所示。该过程中需要用户回答 3 项内容以便生成 3 个文件，如窗口中间的 3 行。

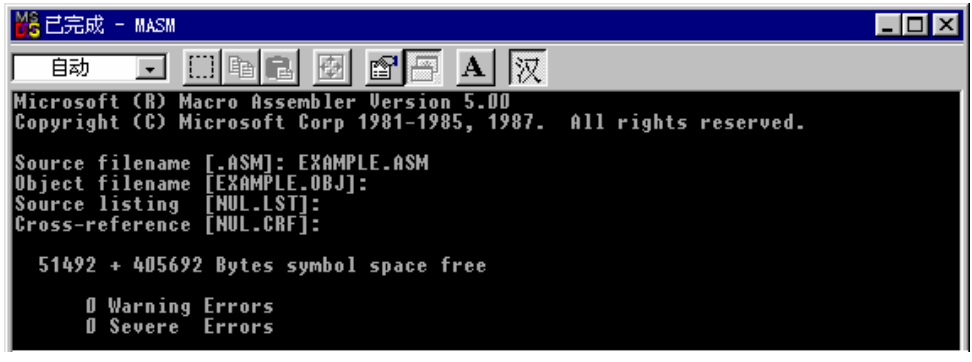


图 4.5　汇编过程的信息显示及输入

① 第 1 行提示用户汇编将生成的“目标文件”，即 EXAMPLE.OBJ 文件。这是汇编的重要文件。按 Enter 键表示同意生成该文件及系统给该文件的命名，也可输入其他文件名。

② 第 2 行提示汇编是否生成“列表文件”（*.LST）。列表文件包含源程序清单和机器语言程序清单，以及有关的符号表，为方便调试而建立的文件。直接回车表示不需要生成列表文件；若需要该文件，则需要输入一个列表文件名。

③ 第 3 行提示汇编是否生成“交叉引用文件”（*.CRF）。该文件包含了源程序中定义的

所有符号，及其在源程序中的行号。直接回车表示不需要生成“交叉引用文件”。

回答上述 3 项提问后，系统开始汇编。当汇编没有发现源程序错误时将提示：

0 Warning Errors（警告错误）

0 Severe Errors（严重错误）

警告错误不会影响连接和执行的，严重错误必须要修改源程序，直到汇编没有错误为止，否则汇编将不生成目标文件。

2. 生成EXE文件

可使用常用的 Microsoft 公司的 LINK 连接程序根据已生成的 OBJ 文件产生对应的 EXE 文件。现以 EXAMPLE.OBJ 文件连接 EXAMPLE.EXE 文件为例说明产生 EXE 文件的过程。具体操作步骤如下。

（1）通过鼠标导航到 LINK.EXE 文件所在的文件夹下；

（2）在 LINK.EXE 文件的位置上双击左键，启动连接程序；

（3）在连接时，用户根据提示输入目标程序文件名 EXAMPLE.OBJ；

（4）根据提示输入其他有关信息。

其操作过程如图 4.6 所示，该过程中同样需要用户回答 3 项内容以便生成 3 个文件，如窗口中间的 3 行。

图 4.6 连接过程的信息显示及输入

① 第 1 行提示用户连接将生成的“可执行文件”，即 EXAMPLE.EXE 文件。这是最终的执行文件。按 Enter 键表示同意生成该文件及系统给该文件的命名，也可输入其他文件名。

② 第 2 行提示连接是否生成“映像文件”（*.MAP）。映像文件包含每个段在内存中的分配情况。直接回车表示不需要生成映像文件；若需要该文件，则需要输入一个映像文件名。

③ 第 3 行提示连接是否生成“库文件”（*.LIB）。该文件包含了源程序中需要用到的库函数。直接回车表示不需要生成“库文件”。

回答上述 3 项提问后，系统开始连接生成可执行文件。当连接没有发现目标程序时将提示有严重错误，无法生成可执行文件。当程序中未定义堆栈段时，将提示有警告性错误：

Warning： No STACK segment

（警告错误：没有堆栈段）

警告性错误不影响可执行文件的生成，也不影响程序的执行。

3. 快速生成可执行文件的方法

若用户只需生成源文件（ASM 文件）、目标文件（OBJ 文件）和可执行文件（EXE 文件），而不要 LST 文件、CRF 文件、MAP 文件和 LIB 文件时，可用下列命令方式来汇编和连接。

（1）单击“开始”菜单 → “运行”。

（2）在对话框中输入 E:\MASM\MASM EXAMPLE；然后单击“确定”按钮，完成汇编工作。运行过程中屏幕显示如图 4.7 所示的“运行”对话框。

（3）连接文件过程与汇编相似，不同的是在对话框中输入 E:\MASM\LINK EXAMPLE；然后单击“确定”按钮。

图 4.7 “运行”对话框

其中“E:\MASM\”指定了 MASM.EXE 和 LINK.EXE 文件所在的磁盘和文件夹。

命令行末的分号告诉系统省略屏幕提示信息，直接使用默认值，从而提高汇编和连接的速度。

4. 程序的执行和调试

在产生可执行文件，即 EXE 文件后就可以直接执行程序。程序的执行可以使用类似于快速生成可执行文件的方法。区别在于，在“运行”对话框中输入的是要执行的程序名。有的程序不包含输入输出指令，CPU 在执行完用户程序后就自动返回到操作系统，用户可能看不到程序的任何执行结果，上述的 EXAMPLE.EXE 就属于此类程序。若要观察程序的执行结果，最常用的方法是运行 DEBUG 程序来检查。

在汇编、连接成功后，程序的执行未必就能成功，这是因为在汇编、连接时，源程序中可能存在的一些逻辑错误往往不能被发现。可使用调试器 DEBUG 对目标程序进行动态调试，在执行过程中观察各寄存器、相关存储单元及标志寄存器的值，跟踪执行情况，判断结果的正确性。EXAMPLE.ASM 经过汇编、连接生成可执行文件 EXAMPLE.EXE 后，可用 DEBUG 对其进行动态调试。首先将待调试的 EXE 文件调入内存，调入“运行”对话框，如图 4.7 所示。

文件调入后，将显示 DEBUG 命令状态提示符“-”，此时可用 R 命令查看或修改寄存器内容，用 U 命令对程序代码进行反汇编，用 D 命令查看数据单元，用 G 命令执行程序，用 T 命令单步逐条执行程序等。DEBUG 中使用的地址和数据默认为十六进制形式。在输入数据时不要加尾标“H”。下面分别介绍上述命令的使用方法，其他的命令使用见附录。

（1）显示存储单元命令 D（Dump）

格式：-d [地址]

-d [地址范围]

功能：显示指定地址或地址范围内存储单元的内容

① 显示代码段存储单元的内容

-d cs:0000 显示代码段中从 0000H 开始的 128 个字节单元的内容，命令输入及显示结果如图 4.8 所示。从 B8 到 21 这部分内容即为程序 EXAMPLE.ASM 编译链接后产生的机器代码在代码段存储的情况，均以十六进制表示。

显示的首行中：

1402：0000 B8 F1 13 8E D8 32 C0 B9-09 00 8D 36 00 00 02 04　…..2…..6…..

1402：0000 表示代码段 CS 的值为 1405H，偏移地址为 0000H。

B8 F1 13 8E D8 32 C0 B9-09 00 8D 36 00 00 02 04 表示代码段中偏移地址 0000H 到 000fH 的 16 个存储单元中存放的数据值，对应关系是 0000H 单元存储的值为 B8H，0001H 单元存储的值为 F1H……，000fH 单元存储的值为 04H。

…..2…..6….. 表示上述存储单元中的数据为 ASCII 码时，在屏幕的显示，如果该数值为控制字符或者不可显示的字符，则以 . 表示。

其他各行显示的内容表示含义与首行相似。

② 按指定地址显示存储单元的内容。

-d ds:0000　显示数据段中从 0000H 开始的 128 个字节单元的内容，命令输入及显示结果如图 4.8 所示。

③ 按指定地址范围显示存储单元的内容。

-d 0000 003f 显示数据段中 0000H 至 003fH 单元的内容，此处可省略“ds:”。命令输入及显示结果如图 4.8 所示。

说明：如果第一次使用 D 命令时，没有指定地址或地址范围，则显示代码段的内容，例如图 4.8 中的第一个 d 命令；其后没有指定地址或地址范围的 D 命令，将在上一次显示内容的基础上，显示其后 128 个单元的内容。

当 D 命令显示代码时，采用的是二进制数的十六进制表示，可读性比较差，无法直接判断出每一个十六进制代码所代表的含义，导致初学者无法直接看出程序的正确与否。这时，可以通过 U 命令将十六进制表示的机器代码转换成为汇编语言助记符。

（2）反汇编命令 U（Unassemble）

格式：-u [地址]

　　-u [地址范围]

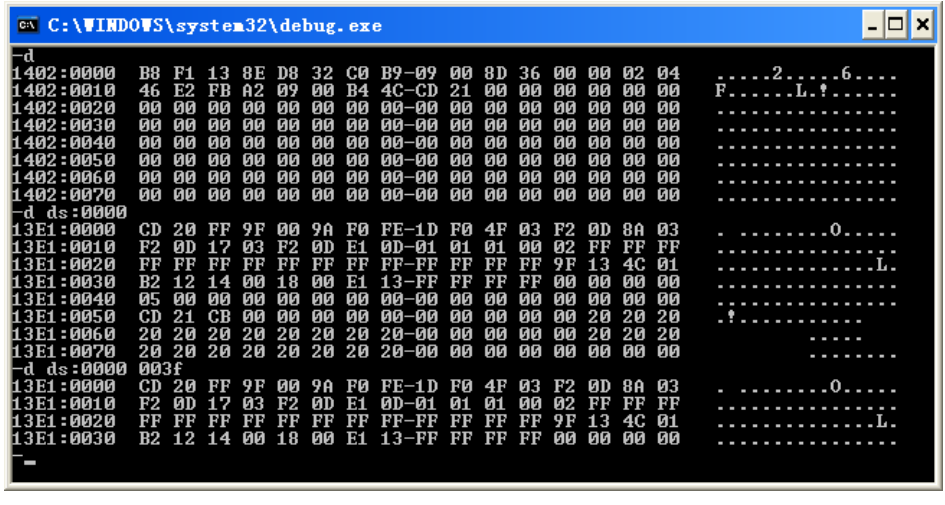


图 4.8　D 命令执行显示结果

功能：对指定地址或地址范围的目标代码进行反汇编，例如

　　-u 0000

从代码段中偏移地址为0000H开始的单元开始反汇编，反汇编显示的结果如图4.10所示。

-u 0000 0018

将地址 0000H 至 0018H 之间的目标代码进行反汇编。

图 4.9 中可以看出，代码段中从偏移地址 0000H 单元开始到 0019H 单元均为有效地数据，从 001AH 单元开始往后都是“00”，也即程序 EXAMPLE.EXE 在由操作系统装入内存后占用的存储空间偏移地址是 0000H～0019H。使用反汇编 U 命令时，将会对目标地址单元的数据反汇编为汇编语言助记符，如图 4.9 显示的。

```
1402: 0000    B8F113     MOV     AX, 13F1
1402: 0003    8ED8       MOV     DS, AX
1402: 0005    32C0       XOR     AL, AL
M
```

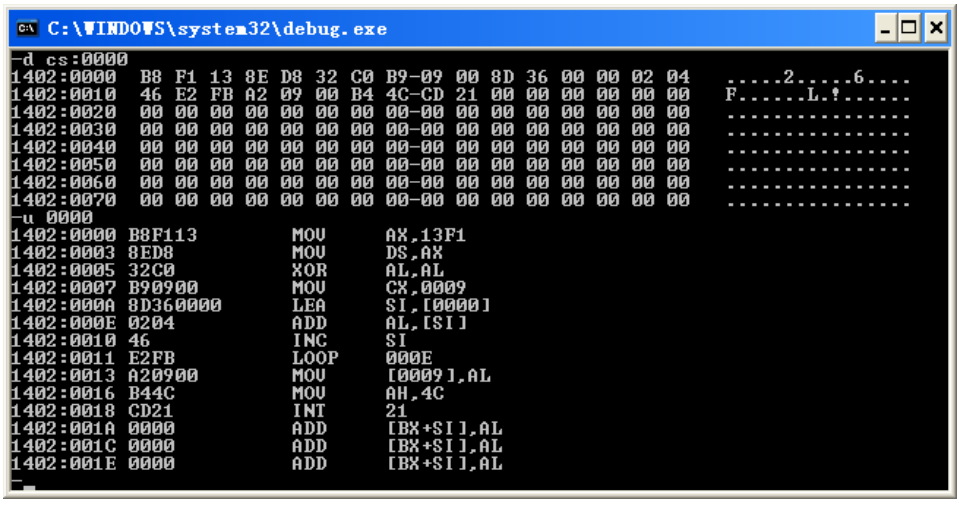


图 4.9　U 命令执行显示结果

从偏移地址 0000H 到 0002H 单元，3 个单元存放的数据为 B8H、F1H、13H，该数据反汇编成汇编语言助记符为 MOV AX, 13F1，这样就很容易知道该指令的作用了；偏移地址 0003 和 0004 单元存放的数据为 8ED8，该数据反汇编成汇编语言助记符为 MOV DS, AX。以此类推，其他反汇编得到的指令在图 4.9 中的显示就比较容易理解了。

注意，在显示中 1402: 001A　0000　ADD [BX+SI], AL，该指令以及其后的指令均无意义，从 D 命令显示的代码段数据中，已经可以看出从偏移地址 001AH 单元开始数据值已经全部为“0”了，所以反汇编得到的汇编语言助记符是没有意义的。

通过以上的 D 命令和 U 命令，已经能够检查 DEBUG 调试器载入的 EXAMPLE.EXE 程序与 EXAMPLE.ASM 之间的对应关系和两者之间的差异了，在汇编语言源程序中的伪指令和其他说明性的语句在 EXAMPLE.EXE 中已经全部转换完成。

在 EXAMPLE.EXE 程序未执行时使用 D 命令查看数据段内容，得到的数据段 DS 寄存器的值为 13E1H（如图 4.8 所示），使用 U 命令反汇编后查看指令时，系统为程序分配的数据段 DS 寄存器的值为 13F1H，同样都是系统分配的数据段，为什么两者相差了 256 个单元呢？

原因是磁盘上的 EXE 文件包括两部分：一部分为装入模块，另一部分为“重定位信息”。DEBUG 调试器在载入 EXAMPLE.EXE 程序到内存时，这两部分均调入内存，在完成对装入

模块的重定位后，重定位信息即被丢弃，然后再在同一内存块的用户程序上方（低地址处）偏移地址为 00H～FFH 的单元自动生成一个有 256 个字节的数据块，该数据块被称为“程序段前缀”（Program Segment Prifix，简称 PSP）。载入程序时，系统自动给 DS，ES，FS 和 GS 赋值，令 DS=ES=存放 PSP 的段基址，FS=GS=0，CS:IP=用户程序的启动地址，然后系统将控制权交给用户程序。

这时，用户必须注意系统给 DS，ES 段寄存器所赋的初值并不等于用户程序数据段、附加数据段的段基址，由于 PSP 的存在，两者相差了 256 个字节，因此用户程序一开始就必须对 DS 进行初始化（程序中如使用 ES 段，也必须初始化）。有关 EXE 文件的内存映像读者可参考其他相关资料。

（3）执行命令 G（Go）

格式：-g [=地址 1][地址 2]

功能：从地址 1 开始执行程序，一直执行到地址 2。

若程序能够正确地执行结束，屏幕将显示当前寄存器的执行结果及后一条将要执行的指令。这时用户可以查看寄存器中的数据变化情况，也可以使用 D 命令查看各个段的信息。用户可以使用两种方法来确定地址 1 和地址 2 的具体值：其一，在汇编时，产生 LST 文件，用文本编辑器查看*.LST 文件；其二，在 DEBUG 中，使用 U 命令查看程序。例如：在 EXAMPLE.ASM 程序中，最后一条指令是 INT 21H，该指令的地址为 0018H，故可以使用命令：

g=0000 0018

命令的执行情况如图 4.10 所示。图中显示了处理器中的通用寄存器 AX, BX, CX, DX, SP, BP, SI, DI 的当前值，该值是执行完指令 MOV AH, 4CH 后寄存器的状况，除此之外，还显示段寄存器 DS, ES, SS, CS 的当前值以及 IP 的值。在本书前面章节中，曾经介绍 IP 寄存器中存放的是将要执行指令的偏移地址，这里 IP=0018H，说明了存储在偏移地址为 0018 单元的指令并未执行，也即 INT 21H 指令并未执行。

图中 4.10 显示的 NV UP EI PL NZ NA PE NC 表示了指令执行后状态标志寄存器的值，各符号所代表的含义如表 4.6 所示。

注意，状态标志寄存器的值是指令执行到 MOV AH, 4CH 后寄存器的状况，如需要查看每条指令执行后寄存器的值，则需要使用 DEBUG 中的 T 命令。

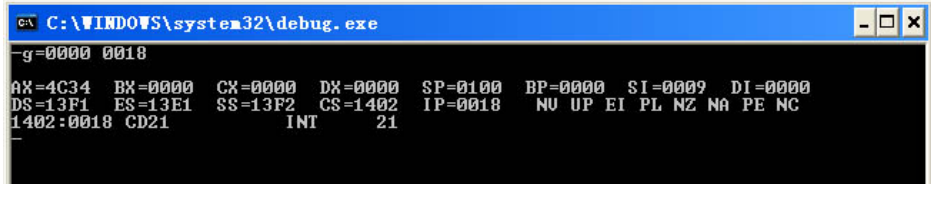


图 4.10　G 命令执行的显示结果

INT 21H 指令执行后将返回操作系统，无法观察指令执行后寄存器及状态标志值的变化。若确需执行该指令，则命令的结束地址应改为 001A，即使用命令：

g=0000 001a

表 4.6　FLAGS 中的状态标志的状态表示符号

标　　志	标志位为 1 的符号表示	标志位为 0 的符号表示
OF	OV	NV
DF	DN	UP
IF	EI	DI
SF	NG	PL
ZF	ZR	NZ
AF	AC	NA
PF	PE	PO
CF	CY	NC

（4）单步执行命令 T（Trace）

格式：-t [指令数]

功能：执行一条或多条指令。每执行一次，则显示所有寄存器内容和状态标志值。

如果不指定指令数，则执行当前 CS:IP 指定地址的一条指令。显示这条指令执行后所有寄存器内容、状态标志值，并显示后一条将要执行的指令。如果指定指令数，则从当前 CS:IP 指定地址开始执行指定数目的指令，并显示每一条指令执行后的结果。T 命令的执行情况如图 4.11 所示。

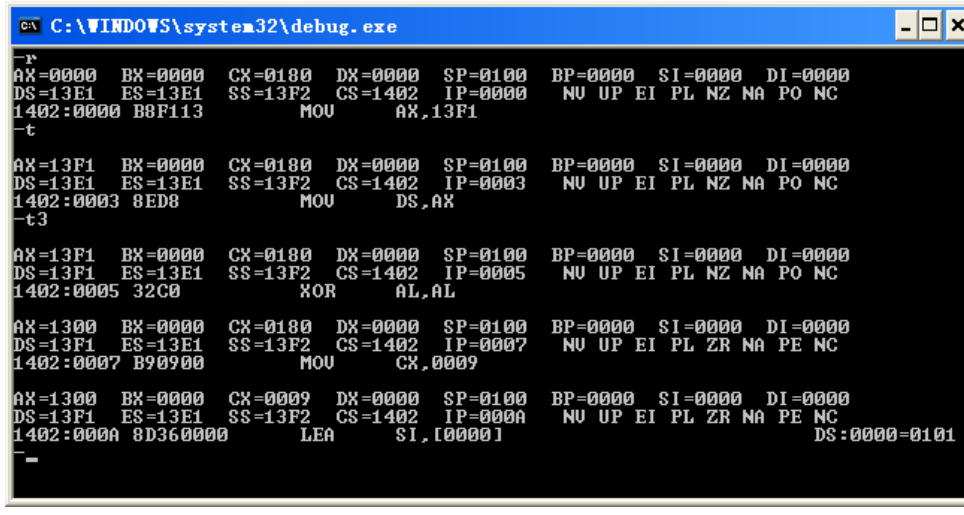


图 4.11　T 命令执行显示结果

首先使用 R 命令使第一条指令 MOV AX，13F1H 显示出来（R 命令将在后面详细介绍），同时能够观察到各寄存器的初始状态。因为 T 命令不带参数将执行一条指令，图中第一个 T 命令执行的就是 MOV AX，13F1H 后各寄存器的状态。通过对比读者可以发现，该指令的执行改变了 AX 寄存器的值，由 0000H 变为 13F1H，其他寄存器的值均无变化，状态标志寄存器同样未发生改变，在本书 3.4.1 节中已经介绍过。

T 命令执行指令时，如果指令中涉及到数据段，则 DEBUG 调试器自动显示该数据段中相应单元的数值。

（5）检查和修改寄存器内容命令 R（Register）

格式：　-r [寄存器]

功能：显示和修改 CPU 中寄存器内容和状态标志值。

该命令可以显示所有寄存器的内容，例如

-r

系统显示结果如图 4.12 所示。该命令也可以显示某个寄存器的内容，例如

-r ax

系统显示结果如图 4.12 所示。即 AX 寄存器的当前内容为 0000H，如不修改则直接按 Enter 键，否则可输入欲修改的内容，例如

-r bx

BX　0000

: 0123　把 BX 寄存器的内容修改为 0123H。系统显示结果如图 4.12 所示。

该命令还可以显示和修改状态标志值，例如

-r f

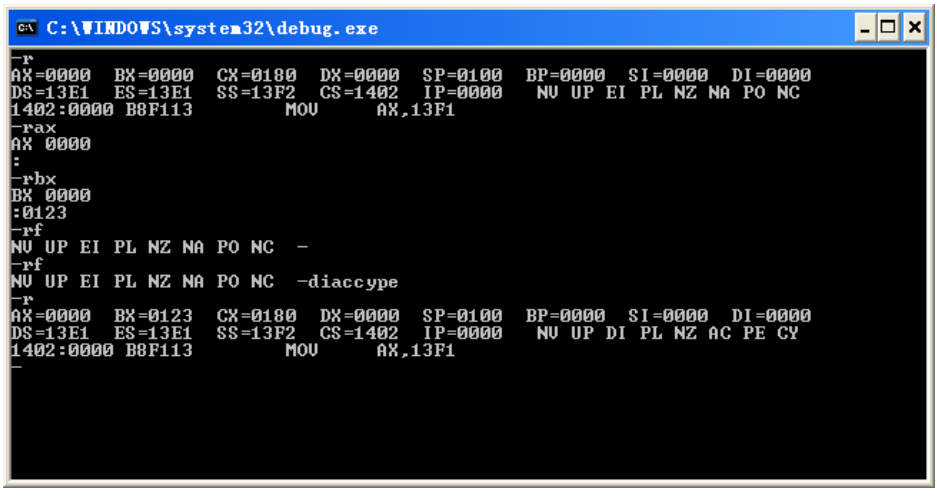


图 4.12　R 命令显示结果

系统显示状态标志值，此时如不修改可按 Enter 键；否则，可输入欲修改的内容，各符号代表的含义如表 4.6 所示。例如，欲修改 IF、AF、CF 及 PF 标志，可在显示的各状态标志值之后输入 diaccype，即

NV　UP　EI　PL　NZ　NA　PO　NC –diaccype

（6）修改存储单元内容命令 E（Enter）

格式 1：-E 地址 [数据表]

功能：用给定的数据表来替代指定范围的存储单元内容。

例如，-E　DS:0010　11 'XYZ' 22

其中 11H，'X'，'Y'，'Z' 和 22H 各占一个字节，该命令可以用这 5 个字节来替代存储单元 DS:0010 到 0014 的原先的内容。指令执行结果如图 4.13 所示。图中 DS:0010 到 0014 的原先的内容为 F2H，0DH，17H，03H，F2H，指令执行后 DS:0010 到 0014 的内容为 11H，

格式 2：-E 地址

功能：逐个单元相继修改存储单元内容。

例如，-E　DS:0015 则可能显示为：

13E1: 0015　0D.

如果需要把该单元的内容修改为 88，则用户可以直接输入 88，再按“空格”键可接着显示下一个单元的内容，如下：

13E1: 0015　0D. 88　　E1.

这样，用户可以不断修改相继单元的内容，直到用 Enter 键结束该命令为止。指令执行结果如图 4.13 所示。

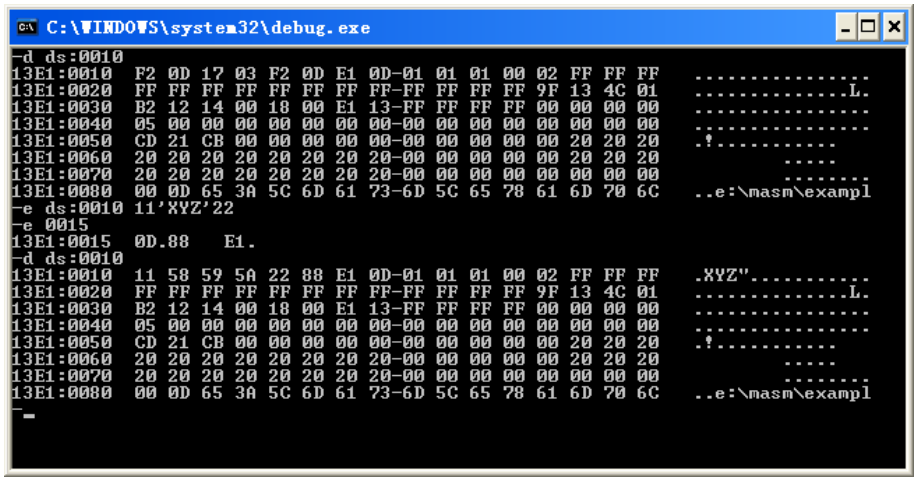


图 4.13　E 命令显示结果

（7）帮助命令 ？

格式 1：　DEBUG/?

功能：显示 DEBUG 调试器装载文件的命令格式。

在系统的命令提示符 C:\>下 DEBUG/?输入 Enter，如图 4.14 所示。

DEBUG [[drive:][path]filename [testfile-parameters]]

DEBUG 调试器可选参数中

[drive:]：表示被装载文件所在的驱动器

[path]：表示被装载文件所在的文件夹

filename：表示被装载文件名称

[testfile-parameters]：表示调试参数

注：[]表示可选项

该提示信息指出，具体 DEBUG 调试器命令可在运行 DEBUG 调试器后输入?得到显示。

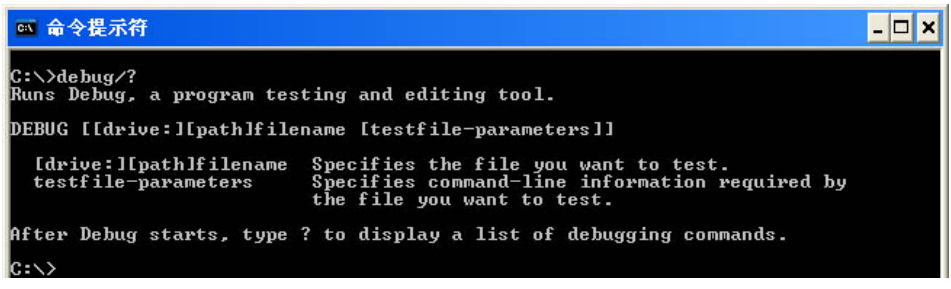


图 4.14　debug 帮助显示结果

格式 2： -?

功能：显示 DEBUG 调试器命令。

该命令可以显示命令的具体内容和参数，例如

-?

系统显示结果如图 4.15 所示。

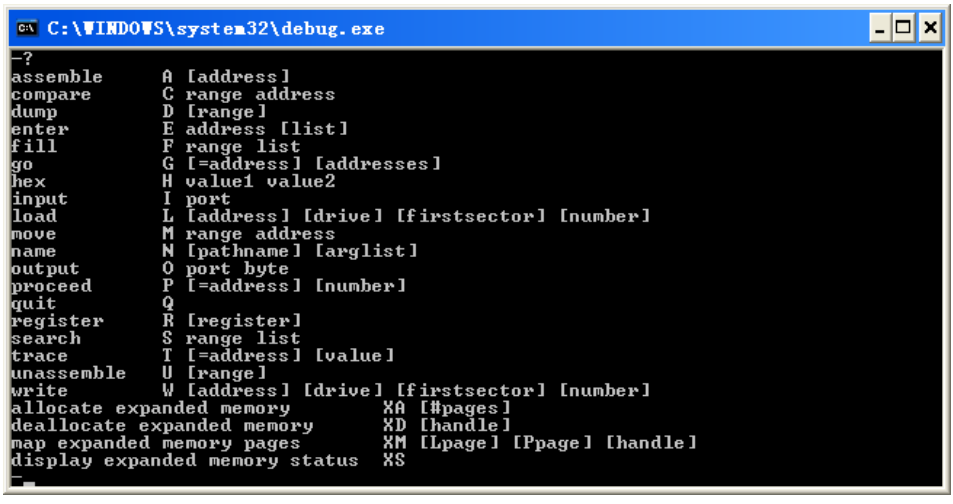


图 4.15　debug 命令帮助显示结果

在 DEBUG 调试器命令中，其他的命令含义为：

compare　C range address ——比较

fill　F range list ——填充

hex　H value1 value2 ——十六进制运算

input　I port ——指定端口输入

load　L [address] [drive] [firstsector] [number] ——加载

move　M range address ——移动

name　N [pathname] [arglist] ——命名

output　O port byte ——输出

proceed　P [=address] [number] ——执行

search　S range list ——搜索

write　W [address] [drive] [firstsector] [number] ——写入

allocate expanded memory　XA [#pages] ——分配扩展内存

deallocate expanded memory　XD [handle] ——释放扩展内存

map expanded memory pages　XM [Lpage] [Ppage] [handle] ——映射扩展内存页

display expanded memory status　XS——显示扩展内存状态

DEBUG 的上述命令，在这里不一一详细讲解，想详细了解这些命令请参考附录 D。

（8）退出命令 Q（Quit）

格式：-q

功能：退出 DEBUG 调试程序，返回操作系统。

DEBUG 调试程序结束后，需要返回操作系统时，在提示符后输入 Q 命令按 Enter 键即可。

4.4.2 TASM汇编环境

如上所述，使用 MASM、LINK 及 DEBUG 可以实现对汇编语言源程序进行处理，事实上，这些工具可以分别以 TASM、TLINK 及 Turbo Debugger 来替代，TASM 是 Borland 公司的汇编工具，使用方法类似于 MASM，但是调试软件 TD（Turbo Debugger）更加简单直观，适合于初学者。其常见版本还可以对使用 80X86、Pentium 指令的源程序进行处理。

1．Turbo Assembler的使用

TASM 是 Borland 公司开发的 3 个汇编工具版本之一，适合于比较小的模块编译工作，编译速度比其他两种要快一些。除此之外，Borland 公司还开发了 TASMX 和 TASM32 两个版本的汇编工具。下面的举例默认 TASM 汇编工具存放在 E 盘的 TASM 文件下。

（1）TASM 指令格式

命令格式为：

TASM [option] source [,object] [,listing] [,xref]

该命令格式与上述 MASM 的格式类似。如果简单输入命令 TASM，就可以得到 TASM 的可选项和参数的说明信息，但是不能进行汇编工作，这一点与 MASM 的人机会话方式有区别。

Source 指定源程序文件，默认的扩展名是 ASM。

Object 指定生成目标文件的标识，可默认。默认时文件名同源程序文件名，扩展名为 OBJ。

Listing 指定生成列表文件的标识，可默认。默认时一般表示不生成列表文件。

xref 指定生成交叉参考文件的标识，可默认。默认时一般表示不生成交叉参考文件。

TASM 允许一次汇编多个文件，此时，在命令中的 Source 选项中将多个文件使用“+”连接每个文件，当然也可以使用 DOS 系统的通配符“*”和“?”来表示多个源文件。例如：

E：\TASM>TASM EXAMPLE1+ EXAMPLE2

E：\TASM> TASM EXAMPLE*ASM

注意：命令行中有下画线的部分为用户输入的信息。

（2）TASM 汇编工具参数 Option

在 TASM 汇编工具中，提供了较多的汇编参数，这些汇编参数为汇编工具 TASM 提供相关的汇编信息。在汇编参数使用时应注意，每个参数都以“/”起始，参数可以连用，也可以单独使用，参数可以紧接在 TASM 之后输入，然后空一格输入源文件名，也可以在命令行的末尾输入。需要了解更全面的汇编参数，可以在“运行”→“CMD”下或者 MS-DOS 方式下输入

E：\TASM>TASM┆MORE

可以分页显示所有参数的功能说明，下面介绍其中几个常用的参数，其他的读者可自行参考参数说明。

/a，/s ——按字母顺序或者源代码段排序。

/c ——在列表文件中产生“交叉引用信息”。

/d ——为原文件中的变量赋值。

例 4.1 中，可为 SUM 赋初值 0:TASM EXAMPLE.ASM /dSUM=0。该参数可同时为多个变量赋值后再进行汇编。

/l　——产生列表文件。与源文件同名，扩展名为 .LST。

/zi　——产生用于调试的含有完整的信息的目标文件。

用户在汇编源程序时可以根据需要可以选择不同的参数项，例如假设 TASM．EXE 文件和用户的源程序 EXAMPLE．ASM 文件都在 E:\TASM 下，则汇编如下：

E：\TASM>TASM　EXAMPLE．ASM　仅生成 EXAMPLE．OBJ 文件，如果源程序扩展名为.ASM，则可以省略。

E：\TASM>TASM　EXAMPL E　功能与上同。

E：\TASM>TASM　EXAMPL E　/L /ZI　生成 EXAMPLE．OBJ 文件和 EXAMPLE．LST 文件，并包含完整调试信息。

汇编的过程及结果如图 4.16 所示。

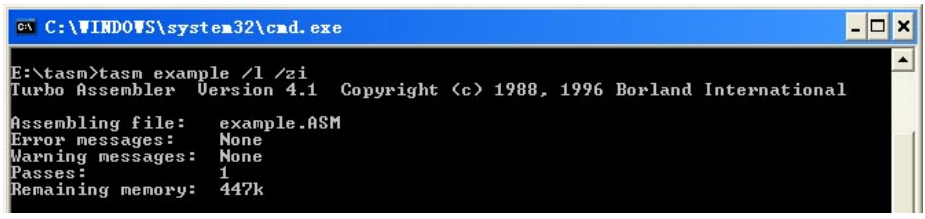


图 4.16　汇编 EXAMPLE 源程序结果

提示信息中，包含了如下信息：

Error　messages：None

Warning　messages：None

表示源程序已通过汇编，没有发现错误。反之，则会显示错误和警告的个数及错误所在的行号和错误类型。

2. Turbo Link的使用

下面介绍的链接器 TLINK 也是 Borland C++的一部分，它的使用方法与 LINK 类似，但是命令选项较多，功能比 LINK 要强，而且它还支持 386 以上的指令，这一点是 LINK 所不具有的功能。命令格式为：

TLINK　objfiles　[,exefile]　[,mapfile]　[,libfiles]

如果简单地输入命令 TLINK，那么就可以得到 TLINK 的可选项和参数的说明信息。

Objfiles 指定欲链接的目标代码文件，默认的扩展名为 OBJ。如果要链接多个目标代码文件，那么文件标识之间用加号或者空格间隔。

Exefile 指定输出的可执行文件名，默认的文件名同第一个目标代码文件名。默认的扩展名一般是 EXE。

Mapfile 指定输出的定位图文件，默认的扩展名是 MAP。默认情况下生成的定位图文件名与可执行文件名相同。

Libfiles 指定链接时使用的库文件，默认的扩展名是.LIB。可以有多个库，间隔同目标代码文件。默认情况下不使用库。

TLINK 链接器中同样提供了很多参数，与 TASM 类似，可使用：

E：\TASM>TLINK　¦ MORE

查看各参数功能。

如果要链接 32 位目标代码（包含 386 以上指令的目标代码），必须使用 32 位方式链接，选用选项 /3。

例如链接 32 位目标代码 EXAMPLE.OBJ 则使用如下命令：

E：\TASM>TLINK　/3　EXAMPLE

链接的过程及结果如图 4.17 所示

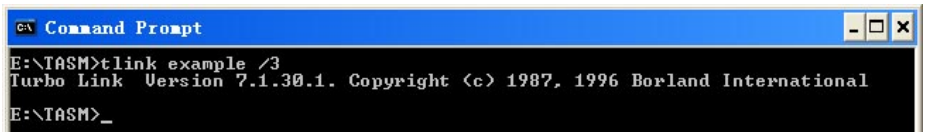


图 4.17　链接 EXAMPLE 目标程序结果

以后各章中带有.386 的源程序均要使用该连接方式。

3. Turbo Debugger的使用

Turbo Debugger 是一个比较先进的源代码级调试器，它可以调试多种语言编写成的程序，而且它还支持 32 位的源代码。为描述方便，将 Turbo Debugger 简称为 TD。

（1）启动 TD 及退出 TD

启动 TD 及装载被调试程序 example.exe 的命令如下：

E：\TASM>TD　EXAMPLE

TD 直接给出如图 4.18 所示的机器指令级调试界面。可能会叠加一个报告无符号表的对话框，按 Esc 键就能关闭该对话框。

退出 TD 比较简单，可以先使用 Esc 键关闭所有对话窗口，然后按下组合键 Alt+X 即可退出 TD，也可以使用主菜单当中 File→Quit 菜单项退出 TD 返回操作系统。

（2）TD 的多窗口界面操作

TD 的机器级调试界面是一个多窗口界面，如图 4.18 所示，中间为 5 个显示调试区，光条停留的区域称之为“活动区”，可以使用 Tab 键在各区之间移动。活动区的快捷键可使用 Ctrl 或者 Alt 激活显示。

① 查看和修改代码区内容

代码区显示的是以行为单位的机器指令和反汇编后的指令助记符。如窗口的第一行：

CS: 0000►B8330B　　mov　ax, 0B33

表示位于代码段偏移地址 0000H 处的 3 个字节的内容：B8330B，它是符号指令 MOV　AX, 0B33 对应的机器指令。其中 B8 是该指令的操作码字段，330B 是操作系统分配给数据段 DSEG 的段基址，所以这条指令就是由源程序中的指令 MOV　AX, DSEG 汇编成的机器指令，然后由 TD 反汇编得到的。窗口的中 CS：0000 后面的“►”符号表示本条指令由 CS：IP 指向，即将执行。

当代码显示区成为活动区时，可进行如下操作：

该区指令的光条可用光标控制键、翻页键调整所显示的代码区。

可使用 Ctrl+G 组合键弹出定位对话框，输入“段寄存器：偏移地址”定位显示。

可使用 Ctrl+S 组合键弹出查找对话框，输入要查找的指令，直接寻找该指令。

组合键 Alt+F10 能够弹出适合本区的操作菜单。

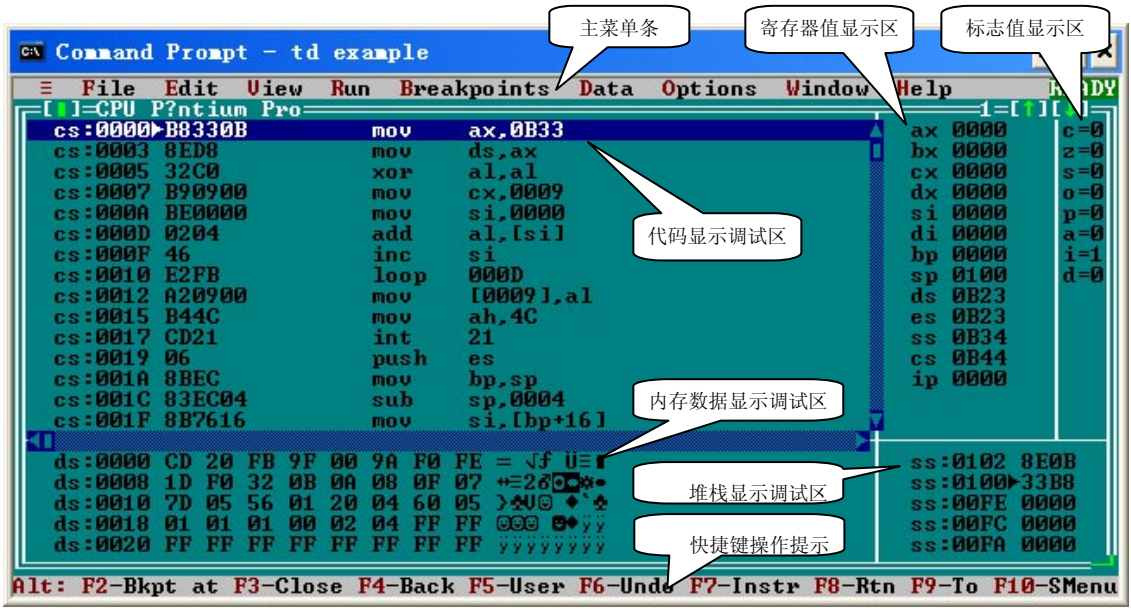


图 4.18　TD 的机器指令级调试界面

② 查看和修改寄存器区内容

该区位于代码区的右侧，按下 Tab 键，光条从代码区移至寄存器区，表示该区为当前活动区。寄存器显示区显示了 CPU 内部各通用寄存器、段寄存器和 IP 的当前值。

该区中可进行 16 位通用寄存器和 32 位通用寄存器的切换，切换有两种方法：

组合键 Alt+F10 弹出菜单，选择其中的“Registers 32-bit”选项。

组合键 Ctrl+R

32 位寄存器显示如图 4.19 所示。

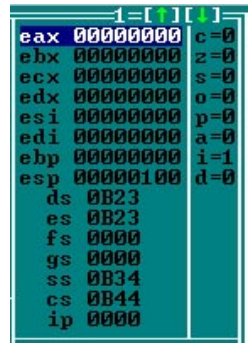


图 4.19　32 位寄存器

修改被光条覆盖的寄存器的内容：

组合键 Ctrl+C，弹出对话框，用户输入有效的数据将替换被光条覆盖的寄存器原有数据。

组合键 Ctrl+Z，被光条覆盖的寄存器清 0。

组合键 Ctrl+I，被光条覆盖的寄存器加 1。

组合键 Ctrl+D，被光条覆盖的寄存器减 1。

③ 查看和修改状态标志位

按下 Tab 键，光条从寄存器区移至状态标志值显示区，当该区成为“活动区”时，使用光标控制键移动光条，组合键 Ctrl+T 可翻转光条所在标志位的状态，即 0 改为 1，1 改为 0。

④ 查看和修改堆栈内容

按下 Tab 键，光条从状态标志值显示移至堆栈显示区，该区显示堆栈栈顶附近个单元的地址和当前值。在图 4.18 中，窗口中 SS：0100 后面的“►”符号表示该位置为当前栈顶，单元的数值为 B833H。

当该区为“活动区”时，按组合键 Ctrl+G 弹出一个用于地址定位的对话框，用户可输入新的堆栈地址定位显示。按组合键 Ctrl+C 弹出输入数值的对话框，输入新的数值可以修改光

标覆盖的堆栈单元的内容。

⑤ 查看和修改内存数据区的内容

该区显示存储器的内容，各列分别显示“段、偏移地址、内容、……、ASCII 码”，最右侧的 ASCII 是由存储单元中十六进制数据转换得到的，所以某些数值对应的 ASCII 码可能是符号或者不显示的控制符。

该区为“活动区”时，可进行下列操作

光标控制键和翻页键可调整窗口显示的内容。

组合键 Ctrl+D 可改变窗口显示内容的格式。

组合键 Ctrl+G 可定位显示存储单元内容（类似于代码区中的组合键功能）。

组合键 Ctrl+S 搜索字节表。

组合键 Ctrl+C 修改存储单元内容。

（3）简单的程序调试过程。

以本章例 4.1 程序 EXAMPLE.ASM 为例，介绍调试的过程。

首先使用 TASM 和 TLINK 完成程序的编译和链接工作，生成 EXE 文件，并启动 TD，加载 EXAMPLE 文件，屏幕出现 EXAMPLE.ASM 源程序，如图 4.18 所示，三角形符号停留在第一条指令上。

① 连续执行程序

按 F9 键（菜单中的 RUN），即可从三角形符号停留的指令上开始执行。

② 查看执行结果

程序中如果有输出到屏幕的指令，则按下组合键 Alt+F5（菜单中的 Windows→User Screen）临时切换到 DOS 屏幕，查看结果，需要返回 TD 的窗口可按任意键。

③ 使光标重新指向启动位置

程序运行结束后，如果用户想重新运行程序，可按下组合键 Ctrl+F2（菜单中的 RUN→Program rest）重新装入程序，并使光条指向第一条指令。

④ 单步执行程序

单步执行程序仅仅执行一条指令，方便用户仔细观察程序运行时各寄存器、存储单元和状态标志位的变化，排除程序的错误。单步执行根据对程序“跟踪”状态的不同，有 3 种单步执行的操作命令。

F8（菜单中的 RUN→Step over）单步操作。执行 CALL 和 INT n 指令时，“不跟踪”相关的子程序。“不跟踪”的含义是指在进入相关子程序后，自动连续执行子程序直至返回。

F7（菜单中的 RUN→Trace into）单步操作。执行 CALL 能够跟踪子程序，随即暂停，等待用户下一步操作。但是遇到 INT n 指令时与 F8 相同操作。

组合键 Alt+ F7（菜单中的 RUN→Instruction trace）单步操作。执行 CALL 和 INT n 指令时进入相关子程序之后立即停止，等待用户操作。

实践证明：用组合键 Alt+ F7 进入服务程序后，如果执行单步操作，很容易造成系统瘫痪，因此请读者慎用。

在上例中，按一次 F7，从寄存器窗口可以看出 AX 寄存器的值改为 0B33H，IP 的值改为 0003H。在 TD 中，每次单步执行后，发生变化的寄存器，标志位都会以高亮显示，便于用户观察。再按一次 F7，从寄存器窗口可以看出 DS 寄存器的值由 0B23H 改为 0B33H，IP 的值改为 0005H，表示程序已经执行完成 5 字节指令。连续按 F7，逐个观察各寄存器、存储

单元的值的变化过程，可以使编程者验证程序的正确性。

以上仅仅是一个简单程序的调试过程，汇编语言程序设计是一项综合性的工作，需要有一定的硬件基础和经验，并结合程序调试，才能更好地理解和学习汇编语言。

习　题

4.1　什么是汇编语言源程序和汇编程序，二者有何区别？

4.2　解释下列名词：

标识符　保留字　伪指令　标号

4.3　下面是一些数据段的定义，画出它们在内存中的分配情况。

（1）数据段 DATA 定义为：

```
DATA   SEGMENT
  V1    DB   70H
  V2     DB    43H
  V3     DW    1200H
  V4     DW    6600H
DATA     ENDS
```

（2）数据段 DATA 定义为：

```
DATA     SEGMENT
  V1    DB   92H, 63H
  V2    DW   8000H
  V3    DB   12H, 2 DUP（01H, 2 DUP（35H））
DATA   ENDS
```

（3）数据段 DATA 定义为

```
DATA   SEGMENT
  V1    DD   12345678H
  V2    DB   'GOOD! '
DATA   ENDS
```

4.4　下面定义的数据段 DATA，假设段地址为 1000H。

```
DATA   SEGMENT
  D1    DB   61H, 52H
  D2    DW   3456H, 0123H
  D3    DB   20 DUP（0）
DATA   ENDS
```

执行下列运算后，填写各寄存器的值。

```
MOV   AX, SEG   D2          AX=_____
MOV   BX, OFFSET D3         BX=_____
MOV   CX, TYPE   D1         CX=_____
MOV   DH SIZE   D2          DH=_____
```

4.5　已经定义了数据段如下：

```
DATA   SEGMENT
    NUM = 48
    X   DB   NUM
    Y   DB   56
    Z   DW   300
DATA   ENDS
```

指出下列指令中的错误

（1）MOV　DS，DATA　　（2）MOV　AX，X

（3）MOV　NUM，BX　　（4）MOV　AL，Z

（4）MOV　Y，BX　　（6）MOV　X，Y

4.6　EQU 与=伪指令的区别是什么？

4.7　下列语句各为变量分配了多少个字节存储单元？

（1）D1　DB　3　　（2）D2　DB　100

（3）D3　DB　'100'　　（4）D4　DW　20, ?, 0, 10 DUP（1）

（5）D5　DW　3, 9, 6　　（6）D6　DD　10, 20

（7）D7　DD　D1　　（8）D8　DW　D5+4

4.8　有以下程序段，试问汇编后符号 N1 和 N2 的值各是多少。

```
STR1　DB　1, 1, 3, 4
STR2　DW　3 DUP（0）, 5
N1　EQU　$-STR2
N2　EQU　STR2-STR1
```

4.9　汇编语言源程序中有哪 4 个段？

4.10　指出汇编、连接过程中能够生成哪 6 种文件，各自的用途是什么。

第 5 章　顺序程序设计

本章在介绍汇编语言程序设计的基本步骤的基础上，讲述顺序程序设计的基本方法，对于顺序程序设计中如何设置伪指令，如何选用指令及其寻址方式，如何安排各条指令的先后顺序进行了举例说明。还介绍了十进制算术运算的相关指令及其使用方法。考虑到程序设计示例中经常需要进行数据的输入和输出，因此本章对常用输入、输出 DOS 系统功能调用也进行了介绍。

5.1　汇编语言程序设计的基本步骤

汇编语言程序设计的基本步骤除具有多种高级语言程序设计的共性外，由于汇编语言本身的特点，在进行汇编语言程序设计时还要考虑数据的属性及其存储区的分配。其基本步骤如下：

（1）分析问题，选择适当的解题方法，并确定解题步骤。

（2）根据具体问题确定原始数据、中间结果及最终结果的数据属性。

（3）对相关数据分配寄存器、存储区。

（4）根据步骤（1）～（3）画出程序流程图。

（5）根据流程图编写汇编语言源程序。

5.2　顺序程序设计

顺序结构是最基本、最简单的一种程序结构。这种结构主要由数据传送及算术逻辑运算指令组成。在这种结构中，各条指令被逐条顺序执行，如图 5.1 所示。

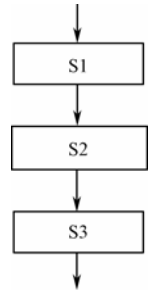


图 5.1　顺序程序结构流程图

在第 3 章的“3.4.2”中已介绍了基本算术运算指令，这些指令针对的是二进制数，即操作数及算术运算结果均是二进制形式。若要进行十进制数的算术运算，则需要在使用算术运算指令的基础上做适当调整。

5.2.1　十进制算术运算

1. BCD码

这里所说的十进制数实际上是十进制数的二进制编码，即 BCD（Binary Coded Decimal）码。具体而言，就是用 4 位二进制数码来表示一位十进制数码。如表 5.1 所示。

表 5.1　十进制数码与 BCD 码对照表

十进制数码	0	1	2	3	4	5	6	7	8	9
BCD 码	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

在 80x86 系统中，允许使用两种 BCD 码：非压缩 BCD 码和压缩 BCD 码。

（1）非压缩 BCD 码

非压缩 BCD 码以 8 个二进制数码表示一个十进制数码，实际上是以低 4 位表示十进制数码，而高 4 位无意义。一个十进制数可表示为一个顺序排列，8 位二进制数码为一组的二进制数字串。为了便于表达，又可表示为一个顺序排列，2 位十六进制数码为一组的十六进制数字串。例如，十进制数 8901 的非压缩 BCD 码为

××××1000　××××1001　××××0000　××××0001B

又可以表示为

×8　×9　×0　×1H

（2）压缩 BCD 码

压缩 BCD 码以 4 个二进制数码表示一个十进制数码。一个十进制数可表示为一个顺序排列，8 位二进制数码为一组的二进制数字串。与非压缩 BCD 码不同，这 8 位二进制数码表示 2 个十进制数码。为了便于表达，又可表示为一个顺序排列，2 位十六进制数码为一组的十六进制数字串。例如，十进制数 8901 的压缩 BCD 码为

1000 1001 0000 0001B

又可以表示为

89 01H

2. BCD码调整指令与十进制算术运算

所谓十进制算术运算，是指参加算术运算的操作数是 BCD 码，结果也是 BCD 码形式。80X86 系统中没有设置专门的十进制算术运算指令，而是通过二进制算术运算指令辅之以 BCD 码调整指令来实现十进制算术运算。BCD 码调整指令如表 5.2 所示。

表 5.2　BCD 码调整指令

分　组		指　令	功　能	对标志位的影响 CF PF AF ZF SF OF	使 用 位 置
非压缩 BCD 码	加法调整	AAA	将 AL 中的和调整为两位非压缩 BCD 码并送 AX	× — × — — —	紧接加法指令之后
	减法调整	AAS	将 AL 中的差调整为两位非压缩 BCD 码并送 AX	× — × — — —	紧接减法指令之后
	乘法调整	AAM	将 AX 中的积调整为两位非压缩 BCD 码并送 AX	— × — × × —	紧接乘法指令之后
	除法调整	AAD	将 AX 中的被除数作调整，以使 DIV 指令执行后得到的商（AL）为非压缩 BCD 码	— × — × × —	作为除法指令的前一条指令
压缩 BCD 码	加法调整	DAA	将 AL 中的和调整为两位压缩 BCD 码并送 AL	× × × × ×—	紧接加法指令之后
	减法调整	DAS	将 AL 中的差调整为两位压缩 BCD 码并送 AL	× × × × × —	紧接减法指令之后

（1）非压缩 BCD 码加法调整指令

格式：AAA

功能：在 AAA 指令前，应该已经使用 ADD、ADC 或 INC 指令，且用 AL 存放二进制加法之和。AAA 指令将 AL 中的和调整为非压缩 BCD 码并送 AX。具体实现方法：若二进制相加后（AL）低 4 位＞9 或 AF=1，则 AL←（AL）+6，AH←（AH）+1，且 AL 高 4 位清零，AF、CF 置 1；否则使 AF、CF 置 0。

【例 5.1】
```
DATA    DB 05H, 08H
          M
        MOV    AH, 0
        MOV    AL, DATA
        ADD    AL, DATA+1        ; （AL）=0DH
        AAA                      ; （AX）=0103H, CF=1
```

【例 5.2】
```
DATA DB 09H, 08H
       M
     MOV    AH, 0
     MOV    AL, DATA
     ADD    AL, DATA+1        ; （AL）=11H
     AAA                      ; （AX）=0107H, CF=1
```

【例 5.3】
```
DATA DB    02H, 06H
       M
     MOV    AH, 0
     MOV    AL, DATA
     ADD    AL, DATA+1        ; （AL）=08H
     AAA                      ; （AX）=0008H, CF=0
```

以上 3 个程序段分别实现了十进制加法运算：

5+8=13

9+8=17

2+6=8

其中的加数为 BCD 数，即十进制数的二进制编码（为方便起见，此处以十六进制形式表示）。ADD 指令实现的是二进制加法运算，而 AAA 指令将二进制加法运算的结果调整为 BCD 码。

（2）非压缩 BCD 码减法调整指令

格式：AAS

功能：在 AAS 指令前，应该已经使用 SUB、SBB 或 DEC 指令，且用 AL 存放二进制减法之差。AAS 指令将 AL 中的差调整为非压缩 BCD 码并送回 AL，向高位的借位在 AH 和 CF 中。具体实现方法是：若 AL 中低 4 位＞9，则 AL←（AL）−6，AH←（AH）−1，且 AL 高 4 位清零，AF、CF 置 1；否则使 AF、CF 置 0。

（3）非压缩 BCD 码乘法调整指令

格式：AAM

功能：在 AAM 指令前，应该已经使用 MUL 指令，且用 AX 存放二进制乘法之积。AAM 指令将 AX 中的积调整为非压缩 BCD 码并送回 AX。

（4）非压缩 BCD 码除法调整指令

格式：AAD

功能：该指令用在除法指令 DIV 之前。该指令对 AX 中的两位非压缩 BCD 码被除数进

行调整，以使其后的 DIV 指令所得到的商（AL）为非压缩 BCD 码。

（5）压缩 BCD 码加法调整指令

格式：DAA

功能：在 DAA 指令前，应该已经使用 ADD、ADC 或 INC 指令，且用 AL 存放二进制加法之和。DAA 指令将 AL 中的和调整为压缩 BCD 码并送回 AL，向高位的进位在 CF 中。

（6）压缩 BCD 码减法调整指令

格式：DAS

功能：在 DAS 指令前，应该已经使用 SUB、SBBA 或 DEC 指令，且用 AL 存放二进制减法之差。DAS 指令将 AL 中的差调整为压缩 BCD 码并送回 AL，向高位的借位在 CF 中。

【例 5.4】
```
DATA DB 42H, 17H
     M
MOV AL,DATA       ; （AL）=42H
SUB AL, DATA+1    ; （AL）=2BH
DAS               ; （AL）=25H, CF=0
```

程序实现了十进制减法运算：

$$42-17=25$$

值得注意的是，被减数和减数的尾标 H 不可少。否则在汇编时存放到 DATA 中的数就不是 42H 和 17H，而是 2AH 和 11H。后者并不是十进制数 42 和 17 的 BCD 码。

5.2.2 汇编语言程序中的输入/输出功能调用

汇编语言的语句与高级语言的语句相比，功能相当简单。用汇编语言编写源程序往往比较困难，解决一个并不复杂的问题的源程序也显得冗长，在涉及到输入/输出管理时尤其是这样。通过在汇编语言源程序中的输入/输出 DOS 功能调用，可以比较方便地实现输入/输出功能，包括从键盘输入字符、将字符显示在屏幕上等功能。

1. 输入/输出DOS功能调用的使用方法

输入/输出 DOS 功能是通过服务子程序来实现的。DOS 对系统功能的服务子程序进行了编号，简称为功能号。某个输入/输出 DOS 功能的调用，实际上就是使程序转向对应的服务子程序。在源程序中进行输入/输出 DOS 功能调用的一般步骤为：

（1）为服务子程序设置入口参数。入口参数的设置一般借助通用寄存器。少数服务子程序无须入口参数，当然就不需要这一步骤。

（2）将功能号送 AH。

（3）使用 INT　21H 指令。

该指令称为软中断指令，其功能是根据 AH 中的功能号转向对应的服务子程序的入口。有些服务子程序执行后还会通过通用寄存器带回出口参数。

2. 常用输入/输出DOS功能调用

（1）01H 号功能

功能：等待从标准设备（如键盘）输入一个字符，将该字符的 ASCII 码送 AL，并在标准输出设备（如屏幕）上显示该字符。

入口参数：无。

调用方式：MOV　AH, 01H

INT　21H

出口参数：AL 的内容为输入字符的 ASCII 码。

说明：当输入的字符为 Ctrl+Break 时，终止程序的执行。

【例 5.5】当指令。

```
MOV　AH, 01H
INT　21H
```

被执行后，等待用户输入字符。若此时用户按下字母键 a，则将字母 a 的 ASCII 码 61H 送 AL，且在显示器上显示字母 a。

（2）02H 号功能

功能：将 DL 中的一个字符显示在标准输出设备（如屏幕）上。

入口参数：DL 的内容设置为待显示字符的 ASCII 码。

调用方式：MOV　DL, 待显示字符的 ASCII 码

MOV　AH, 02H

INT　21H

出口参数：无。

说明：当 DL 中的字符为 Ctrl+Break 时，终止程序的执行。

（3）07H 号功能

功能：等待标准输入设备（如键盘）输入一个字符,将该字符的 ASCII 码送 AL。

入口参数：无。

调用方式：MOV　AH , 07H

INT　21H

出口参数：AL 的内容为输入字符的 ASCII 码。

说明：该功能与 01H 号功能类似，区别有二：一是不显示输入的字符，二是当输入的字符为 Ctrl+Break 时，并不终止程序的执行。不显示输入的字符这一特点适用于输入保密用的口令。

（4）08H 号功能

该功能与 07H 号功能类似，区别仅在于，当输入的字符为 Ctrl+Break 时会终止程序的执行。

（5）09H 号功能

功能：在标准输出设备上显示某个字符串。

入口参数：DX 的内容为要显示的字符串的首地址。

调用方式：MOV　DX, 要显示的字符串的首地址

MOV　AH, 09H

INT　21H

出口参数：无。

说明：确切地说，DX 的内容应该是要显示的字符串的首地址的偏移地址，段地址由 DS 提供。要显示的字符串必须以‘$’作为结束标志。

【例 5.6】以下程序段用以显示信息“Press any key when you ready.”。在用户按下任一键后，另起一行显示信息“Input your password:”。

```
NAME    EXAMPLE5_6
DSEG    SEGMENT
MESS1    DB 'Press any key when you ready', 0AH,0DH, '$'
MESS2    DB 'Input your password: ',OAH,ODH, '$'
DSEG     ENDS
;
SSEG     SEGMENT    STACK
         DB    80H    DUP（0）
SSEG     ENDS
;
CSEG     SEGMENT
         ASSUME    DS: DSEG, SS: SSEG, CS: CSEG
START: MOV    AX, DSEG
         MOV    DS, AX
         ;
         MOV    DX, OFFSET    MESS1
         MOV    AH, 09H
         INT    21H        ; 显示"press any key when you ready. "
         ;
         MOV    AH,08H
         INT    21H        ; 等待用户按任一键
         ;
         MOV    DX,OFFSET MESS2
         MOV    AH,09H
         INT    21H        ; 显示"Input your password:"
         …
```

以上定义变量 MISS1 的伪指令中使用了数据 0AH、0DH，它们是两个特殊字符—“换行”和“回车”所对应的 ASCII 码。显示这两个字符的结果是产生“换行”和“回车”效果，使其后的显示内容从下一行左端开始。

（6）OAH 号功能

功能：将标准输入设备（如键盘）上输入的一串字符送到指定的内存缓冲区。

入口参数：DX 放有内存缓冲区的首地址。

调用方式：

```
MOV    DX, 内存缓冲区的首地址
MOV    AH, 0AH
INT    21H
```

出口参数：无。

说明：确切地说，DX 放有内存缓冲区首地址的偏移地址，段地址由 DS 提供。输入的一串字符以回车符作为结束标志。输入的各字符按地址由小到大的次序存入 DS: DX 指定的内存缓冲区。

值得注意的是，在该功能调用之前，应该在存储缓冲区的第一个字节设置待接受字符的

个数（范围为 1～255）。存储缓冲区的第二个字节在该功能调用后被自动置上实际输入的字符数（回车符除外）。输入的字符从该存储缓冲区的第三个字节开始存放。若实际输入的字符数少于所设置的待接受字符的个数，则多余字节内容被置为零；若实际输入的字符数多于所设置的待接受字符的个数，则多余部分被忽略。

（7）4CH 号功能

功能：返回 DOS。

入口参数：无。

调用方式：MOV　AH, 4CH

　　INT　21H

出口参数：无。

说明：该功能调用常用于汇编源程序代码段的末尾处，使程序返回 DOS。若需要用 DEBUG 等调试工具检查程序执行结果，如检查某些寄存器或内存单元的内容，则应该在程序执行至该功能调用之前进行。

5.3　顺序程序设计综合举例

【例 5.7】求图 5.2 所示阴影部分的面积。

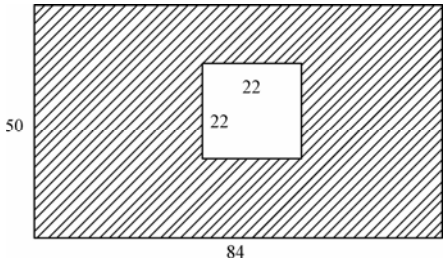


图 5.2

解决这一问题的步骤如下:

（1）建立数学模型

$$S=a\times b-c^2$$

根据算术运算规则可知，应先计算 c^2，然后计算 $a\times b$，最后做减法运算。

（2）原始数据为 a=84，b=50，c=22，可以视为字节数。中间结果 c^2 及 $a\times b$ 超出了字节数表示范围，为 2 字节数。最终结果为这两个中间结果之差，亦为 2 字节数。

（3）原始数据一般用 DB、DW 或 DD 等伪指令定义。对于本题而言，应选用 DB 伪指令定义 a、b 及 c 这 3 个字节变量。由于中间结果 c^2 及 $a\times b$ 为 2 字节数，故可以用 2 字节的存储单元或通用寄存器存放。若用存储单元，则应预先使用 DW 伪指令定义 2 字节变量；若用通用寄存器，则可在 AX、BX、CX、DX、SP、BP、SI 或 DI 中任选，但应避免使用那些存有待使用数据的寄存器。最终结果可以存放于存储单元或通用寄存器。对于本题而言，最终结果宜用 2 字节存储单元存放，故用 DW 伪指令定义字变量 S。

（4）在使用 8086/8088 指令集的情况下，由于 c^2、$a\times b$ 的计算均要使用到寄存器 AX，且接着要进行 $a\times b-c^2$ 的减法运算，从而决定了宜首先计算 c^2，且将寄存器 AX 中的 c^2 值暂存至另一通用寄存器中，然后计算 $a\times b$，最终将 AX 中的 $a\times b$ 值减去暂存的 c^2 值。程序流程图如图 5.2 所示。

（5）根据程序流程图 5.3，可编写出程序如下：

```
NAME    EXAMPLE5_7
DSEG    SEGMENT
A       DB 84
B       DB 50
```

```
C          DB 22
S          DW ?
DSEG    ENDS
 ;
SSEG    SEGMENT   STACK
          DB   80H   DUP（0）
SSEG    ENDS
;
CSEG   SEGMENT
          ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START: MOV   AX, DSEG
          MOV   DS, AX
          ;
          MOV   AL, C
          MUL   AL              ; AX←c²
          MOV   DX, AX
          MOV   AL, A
          MUL   B               ; AX← a×b
          SUB   AX, DX          ; AX← a×b–c²
          MOV   S,   AX
          ;
          MOV   AH, 4CH
          INT   21H             ; 返回 DOS
CSEG   ENDS
          END START
```

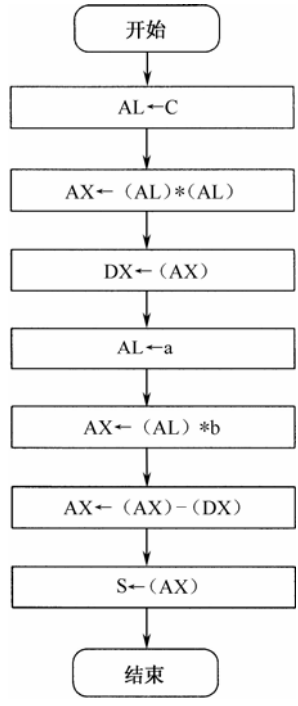


图 5.3　例 5.7 程序流程图

对本例所提出的问题的处理体现了汇编语言程序设计的基本步骤。需要指出的是，并非每一个问题的处理都必须严格遵循这些步骤，可以根据具体情况，在步骤上作适当增减。

【例 5.8】根据键盘输入的 0～9 中任一数字，计算出该数字的平方值，并以非压缩 BCD 码的形式存放于 AX。AH 存放十位数，而 AL 存放个位数。

分析：当从键盘输入 0～9 中任一数字，AL 中存放的实际上是‘0’～‘9’中的任一字符，也即（AL）是 30H～39H 中的任一个 ASCII 码。只要将 AL 高 4 位清零而保持其低 4 位不变，即可得到对应数字的非压缩 BCD 码。将（AL）既作为被乘数又作为乘数，使用乘法运算指令即可得到输入数字的平方值（AX）。最后进行 BCD 调整就可以在 AX 中得到本题指定形式的平方值，程序流程图如图 5.4 所示。程序如下：

```
NAME   EXAMPLE5_8
SSEG    SEGMENT   STACK
          DB   80H   DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
```

```
        ASSUME   SS: SSEG, CS: CSEG
START: MOV   AH, 1
       INT   21H ; AL ←输入数字字符, 如 '8';
       AND   AL, 0FH
       MUL   AL   ; AX←8*8 即 40H;
       AAM        ; AX←0604H ;
       ;
       MOV   AH, 4CH
       INT   21H ; 返回 DOS
CSEG   ENDS
       END   START
```

【例 5.9】求任一给定自然数 N（1≤N≤40）的立方值，并将立方值送变量 CUBE。

分析：自然数 N（1≤N≤40）的立方值不超过 64000，小于 2 字节无符号数的表示范围，故可以使用 DW 伪指令建立一个自然数的立方表，将表中第 N 个元素设置为 N^3。在基址寄存器 BX 设置该表的首地址，将 2N 作为位移量送 SI（因为表中每个元素占 2 字节），经基址变址寻址即可得到 N^3。程序流程图如图 5.5 所示。程序如下：

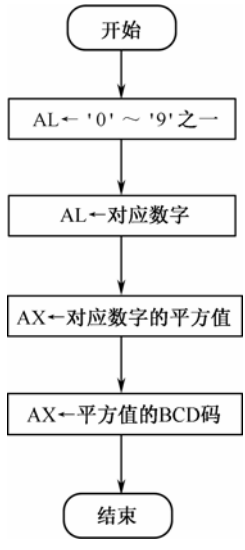


图 5.4　例 5.8 程序流程图

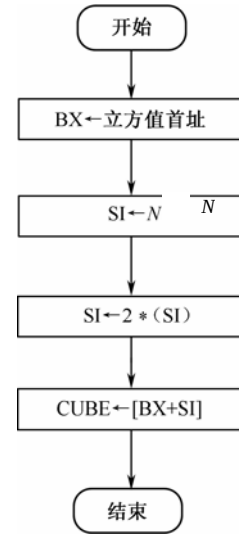


图 5.5　例 5.9 程序流程图

```
DSEG    SEGMENT
CUBTAB  DW   0, 1, 8, 27, … , 64000
N       DB   (1～40 中的任一自然数)
CUBE    DW ?
DSEG    ENDS
;
SSEG    SEGMENT   STACK
        DB   80H   DUP(0)
SSEG    ENDS
```

```
;
CSEG    SEGMENT
        ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START: MOV   AX, DSEG
        MOV   DS, AX
        MOV   BX, OFFSET CUBTAB; BX 指向立方表首址
        ;
        MOV   AL, N
        XOR   AH, AH   ; AX←N
        ;
        MOV   SI, AX
        SHL   SI, 1          ; SI←N3 的存放位置相对立方表首址的位移量
        MOV   AX, [BX+SI]
        MOV   CUBE, AX
        MOV   AH, 4CH
        INT   21H
CSEG    ENDS
        END   START
```

【例 5.10】把变量 X 中的一位十六进制数显示在屏幕上。

分析：根据题意，变量 X 的内容为 00H～0FH 之一，要求显示对应的字符，即 '0'，'1'…'9'，'A'，…，'F' 之一。显示字符可以通过 02H 号功能调用实现，于是应将十六进制数对应的字符（即 ASCII 码）送 DL。对于十六进制数 00H～09H，其对应字符的 ASCII 码为 30H～39H，也就是说将这些十六进制数加上 30H 即可得到对应字符的 ASCII 码；而对于十六进制数 0AH～0FH，其对应字符的 ASCII 码为 41H～46H，也就是说将这些十六进制数加上 30H 后还要再加上 07H 方能得到对应字符的 ASCII 码。这里使用 80386 指令编程，程序如下：

```
.386（或.486、.586）; 选择 80386（或 80486、Pentium）指令集
DSEG     SEGMENT
X        DB   （00H～0FH 之一）
DSEG     ENDS
;
SSEG     SEGMENT   STACK
         DB   80H DUP（0）
SSEG     ENDS
;
CSEG     SEGMENT USE16; 16 位段
          ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START: MOV   EAX, DSEG
       MOV   DS, AX
       MOV   AL, X
        ADD   AL, 30H
```

```
        ;
        CMP   AL, '9'
        SETA  DL      ;  （AL）＞'9'则 DL←1, 否则 DL←0
        MOVZX  DX, DL; （DL）扩展至 DX
        IMUL  DX, 7
        ADD   AL, DL  ;  （AL）为一位十六进制数对应的字符
        ;
        MOV   AH, 02H
        INT   21H; 显示字符
          ;
         MOV   AH, 4CH
         INT   21H; 返回 DOS
CSEG    ENDS
        END   START
```

习　　题

5.1　根据给定指令填空。

（1）MOV　AL, 06H
　　SUB　AL, 08H
　　AAS
　　（AL）=____ CF=_____

（2）MOV　AX, 0605H
　　MOV　BL, 08H
　　AAD
　　DIV BL
　　（AL）=____

（3）MOV　AL, 07H
　　MOV　BL, 06H
　　MUL　BL
　　AAM
　　（AX）=____

（4）MOV　AL, 85H
　　ADD　AL, 29H
　　DAA
　　（AL）=____ CF=___

5.2　以下程序的功能是：首先显示提示信息“Input your password please:”；然后等待用户输入两位字符的口令，口令中的两个字符存储到变量 PASSW 中但不显示在屏幕上，用户每按一键则屏幕显示一个“*”，两个“**”显示在提示信息的下一行。试填写方框中的指令。

```
DSEG    SEGMENT
PASSW   DB ?, ?
DISP    DB 'Input your password please: ', 0AH, 0DH, '$'
MM      DB '*$'
DSEG    ENDS
;
SSEG    SEGMENT  STACK
        DB  80H  DUP（0）
SSEG    ENDS
```

```
;
CSEG    SEGMENT
        ASSUME  DS: DSEG, SS: SSEG; CS: CSEG
START: MOV  AX, DSEG
        MOV  DS, AX
        MOV  DX,OFFSET  DISP
             [          ]
        INT  21H
        MOV  SI, OFFSET  PASSW
             [          ]
        INT  21H
        MOV  [SI], AL
        MOV  DX, OFFSET  MM
        MOV  AH, 09H
        INT  21H
        INC  SI
             [          ]
        INT  21H
        MOV  [SI], AL
        MOV  DX, OFFSET  MM
        MOV  AH, 09H
        INT  21H
        MOV  AH, 4CH
        INT  21H
CSEG    ENDS
        END  START
```

5.3　以下程序的功能是计算表达式 $S=(23000-(X\times Y+Z))/Z$ 的值。试填写方框中的指令。

```
DSEG    SEGMENT
X       DW 600
Y       DW 25
Z       DW –2000
S       DW ?, ?
DSEG    ENDS
;
SSEG    SEGMENT  STACK
        DB  80H  DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
```

```
                ASSUME    DS: DSEG, SS: SSEG, CS: CSEG
START:          MOV    AX, DSEG
                MOV    DS, AX
                MOV    AX, X
                ┌──────────┐
                │          │
                └──────────┘
                MOV    BX, AX
                MOV    CX, DX
                MOV    AX, Z
                ┌──────────┐
                │          │
                └──────────┘
                ADD    BX, AX
                ADC    CX, DX
                MOV    AX, 23000
                CWD
                SUB    AX, BX
                ┌──────────┐
                │          │
                └──────────┘
                IDIV   Z
                MOV    S, AX
                MOV    S+2, DX
                MOV    AH, 4CH
                INT    21H
CSEG            ENDS
                END    START
```

5.4　编写计算 $d=b^2-4ac$ 的程序，设 $a=2$，$b=9$，$c=8$。要求在屏幕上以十进制形式显示结果（提示：借助 BCD 码调整指令实现十进制算术运算；将得出的 BCD 码结果转换为对应的 ASCII 码；通过输入/输出 DOS 功能调用显示 ASCII 码对应的字符）。

第 6 章　分支程序设计

在解决实际问题时，经常需要针对不同情况做出不同的处理。解决这种问题的程序仅采用顺序程序结构还不够，还需要采用分支程序结构。本章在介绍各种分支结构，讲述构成分支的转移指令之后，着重讲解分支程序设计的方法和常用技巧。

6.1　分支程序结构

分支程序结构是指根据条件转向不同程序分支的结构，具体可分为二分支程序结构（相当于高级语言中的 IF-THEN-ELSE 或 IF-THEN 结构）和多分支结构（相当于高级语言中的 CASE 结构）。二分支程序结构通常根据某一条件成立与否确定转向两个程序分支之一；而多分支程序结构通常根据某一条件的多个取值确定转向相应的程序分支，多分支程序结构亦可由多个二分支程序结构构成。

分支程序结构如图 6.1 所示。其中图（a）、（b）为二分支程序结构，图（c）为多分支程序结构，而图（d）为多个二分支程序结构构成的多分支程序结构。

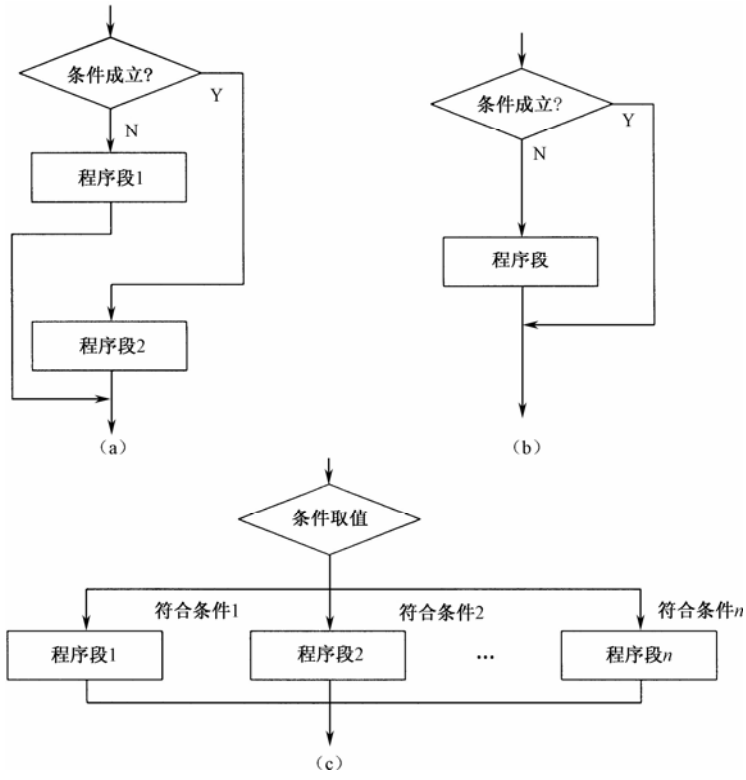


图 6.1　分支程序结构示意图

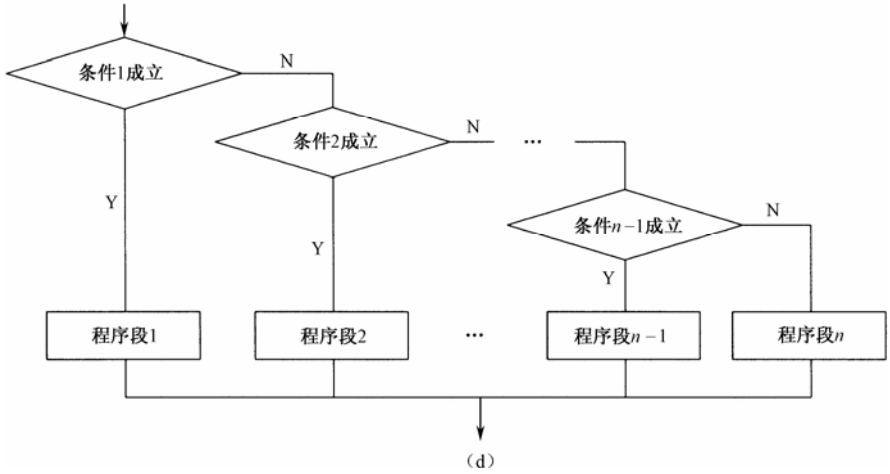


图 6.1　分支程序结构示意图（续）

6.2　转移指令

进行分支程序设计需要使用转移指令。80x86 及 Pentium 指令集中的转移指令分为条件转移指令和无条件转移指令两大类。

6.2.1　条件转移指令

条件转移指令的一般格式为:

JXX　标号

条件转移指令使 CPU 根据一个或两个状态标志的状况确定是转向标号指定的目的位置,还是顺序执行。使用该类指令可以构成二分支程序结构。显然，在条件转移指令之前必须设有影响状态标志的指令。

例如：实现 AL 与 80H 比较的指令

CMP　AL, 80H

根据 AL-80H 的值影响进（借）位标志 CF、偶标志 PF、半进（借）位标志 AF、零标志 ZF, 负号标志 SF 及溢出标志 OF。其后的条件转移指令则可根据比较结果确定是否转移。条件转移指令如表 6.1 所示。

表 6.1　条件转移指令

分　类	助 记 符	功　能	判 断 条 件	备　注
单标志条件转移指令	JC	有进（借）位则转	CF=1	JX 与 JNX 指令的判断条件互为相反
	JNC	无进（借）位则转	CF=0	
	JP	“1”个数为偶则转	PF=1	
	JNP	“1”个数为奇则转	PF=0	
	JZ	为零则转	ZF=1	
	JNZ	非零则转	ZF=0	
	JS	为负数则转	SF=1	
	JNS	为正数则转	SF=0	
	JO	溢出则转	OF=1	
	JNO	未溢出则转	OF=0	

续表

分　类	助记符	功　能	判断条件	备　注
无符号数专用条件转移指令	JA	高于（above）则转	CF=0 且 ZF=0	JA 与 JBE
	JBE	低于等于（below or equal）则转	CF=1 或 ZF=1	JB 与 JAE 的
	JB	低于（below）则转	CF=1	判断条件
	JAE	高于等于（above or equal）则转	CF=0	互为相反
有符号数专用条件转移指令	JG	大于（greater）则转	SF=OF 且 ZF=0	JG 与 JLE
	JLE	小于等于（less or equal）则转	SF≠OF 或 ZF=1	JL 与 JGE 的
	JL	小于（less）则转	SF≠OF 且 ZF=0	判断条件
	JGE	大于等于（greater or equal）则转	SF=OF 或 ZF=1	互为相反

1. 单标志条件转移指令

该类指令根据前一影响标志的指令执行后结果是否为零，有没有进（借）位，有没有溢出等情况确定程序流向。

【例 6.1】设 AL=76H

```
        CMP   AL, 80H
        JNZ   LAB2          ; AL−80H≠0 的条件成立, 即 ZF=0, 转 LAB2 处
LAB1: …
        M
LAB2: …
```

【例 6.2】设 BL=38H。

```
        SHL BL, 1; BL 左移一位, D7 移入 CF
        JC  LAB2;  CF=1 的条件不成立, 顺序执行 LAB1 处的指令
LAB1: …
        M
LAB2: …
```

2. 无符号数专用条件转移指令

该类指令根据两个无符号数的比较结果确定程序的流向。

【例 6.3】设 AL=76H。

```
        CMP   AL , 80H
        JB    LAB2; 无符号数 76H 低于无符号数 80H 的条件成立, 转 LAB2 处
LAB1: …
        M
LAB2: …
        …
```

【例 6.4】设 AL =76H。

```
        CMP AL, 80H
        JAE  LAB1; 无符号数 76H 高于等于无符号数 80H 的条件不成立, 故顺序执行 LAB2 处的指令
LAB2: …
        M
LAB1:  …
        …
```

由【例 6.3】和【例 6.4】可见，当换用判断条件相反的另一条件转移指令时，为使程序功能不变，其后的两段分支程序的位置应该对调。

3. 有符号数专用条件转移指令

该类指令根据两个有符号数（用补码表示）的比较结果确定程序的流向。

【例 6.5】设 AL =76H

```
        CMP   AL , 80H
        JL    LAB2; 有符号数 76H 小于有符号数 80H 的条件不成立, 故顺序执行 LAB1 处的指令
LAB1: …
        M
LAB2: …
```

有符号数大小的比较标准与无符号数不同。对于无符号数而言，每一位均为数值位。对于有符号数而言，在机器中以补码表示，其最高位为符号位。当最高位为“0”，即为正数时，其余位表示数值，当最高位为“1”，即为负数时，所有位取反且末位加“1”的结果方为数值。以单字节数为例，无符号数与有符号数各自的大小关系如图 6.2 所示。

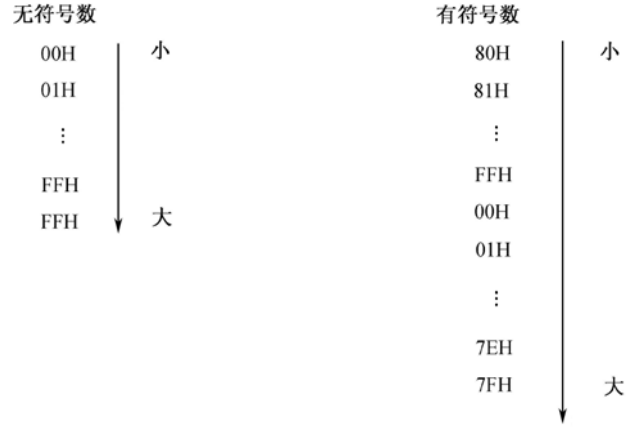


图 6.2

4. 条件转移指令中标号的类型

对于条件转移指令指定的标号存在这样的限制：

（1）对于 80386 以上指令集而言，该标号只能是近标号，即该标号与条件转移指令须处于同一代码段，这种转移称为近转移。

（2）对于 8086/8088 或 80286 指令集而言，该标号只能是短标号，即该标号与条件转移指令不仅须处于同一代码段，而且相距的字节数需在–128～+127 范围内，这种转移称为短转移。与后面介绍的无条件短转移不同的是，无须在标号前加短标号的标识符 SHORT。当实际需要的转移目的位置超出上述限制时，就必须结合使用无条件转移指令。

6.2.2 无条件转移指令

无条件转移指令的一般格式为

JMP 目的位置

无条件转移指令的功能是使 CPU 转向目的位置。目的位置有五种形式，对应 3 种无条件转移指令，如表 6.2 所示。

表 6.2 无条件转移指令

指 令	功 能
JMP SHORT 标号	转移到指定标号处，该标号与该指令不仅须处于同一代码段，而且相距的字节数须在－128～+127 范围内，即实现短转移
JMP 近标号	转移到指定标号处，该标号与该指令须处于同一代码段，即实现直接近转移
JMP FAR PTR 远标号	转移到指定标号处，该标号与该指令不处于同一代码段，即实现直接远转移
JMP MEM16/REG16	由 16 位存储器操作数或 16 位寄存器操作数间接给出目的位置的偏移地址，实现间接近转移
JMP MEM32	由 32 位存储器操作数间接给出目的位置的偏移地址和段地址，实现间接远转移
JMP REG32	此为 80386 以上指令集的指令，由 32 位寄存器操作数间接给出目的位置的偏移地址，实现间接近转移

无条件转移指令通常用于以下 3 种场合：

（1）配合条件转移指令实现条件远转移。

【例 6.6】在【例 6.3】中，若条件转移指令指定的标号 LAB2 与该条件转移指令不在同一代码段，程序将出错。为了解决这个问题，可将该程序段改为：

```
        CMP   AL , 80H
        JAE   LAB1
        JMP   LAB2
LAB1: …
        M
LAB2: …
```

（2）避免一个程序分支“滑入”另一个程序分支。

对于图 6.1（a）所示的二分支程序结构而言，在程序段 1 结束处若不使用无条件转移指令，则将顺序进入，即“滑入”程序段 2，从而造成错误。

【例 6.7】将有符号数 X 和 Y 中的较大者送变量 Z，试写出程序段。流程图如图 6.3 所示。

考虑以下程序：

```
X       DB     60H
Y       DB     94H
Z       DB      ?
        …
        MOV     AL , X
        CMP    AL , Y
          JL    YG    ; AL<Y 则转 YG 处
```

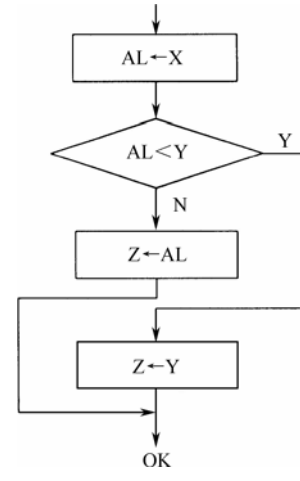


图 6.3 例 6.7 程序流程图

```
        MOV   Z , AL
YG: MOV   BL , Y
        MOV   Z , BL
OK:         …
```

按照题意以及程序流程图，在程序分支

```
        MOV    Z , AL
```

结束后，应转移到 OK 处。但是，上述程序段的实际效果却是在该程序分支结束后“滑入”另一分支，从而造成错误。正确的做法是，在该程序分支结束处，即在指令

```
        MOV    Z , AL
```

后添加指令

```
        JMP     SHORT   OK
```

（3）实现多分支程序结构。

无条件转移指令指定的目的位置可以是存储器操作数或寄存器操作数形式，不同的存储器操作数或寄存器操作数的内容对应着不同的转移目的位置，因此无条件转移指令可以实现多分支程序结构。

【例 6.8】设有以下程序段:

```
  BASE     DW   ZERO, ONE, TWO,  …, N
           …
           JMP   BASE [BX]
           …
ZERO:      …
           JMP   OK
ONE:       …
           JMP   OK
TWO:       …
           JMP   OK
           …
N:         …
           …
OK:        …
           …
```

当 BX=0，2，4…，2N 时，指令 JMP BASE[BX] 分别使程序转向 ZERO，ONE，TWO，…，N 处，如图 6.4 所示。

【例 6.9】设 CS=6032H

```
JMP   CX   ; 转向 6032: CX, 即根据 CX 的不同取值实现多分支转移。
JMP   ECX  ; 此为 80386 以上指令集指令。转向 6032: ECX，即根据 ECX 不同取值实现多分支
             转移。
```

说明：

（1）近转移的实质在于，改变 IP 或 EIP（80386 以上 CPU）总是指向当前代码段中下一顺序指令的规律，而指向目的位置处的指令，即把目的位置的偏移地址送 IP 或 EIP。远转移

不仅要完成这一工作，而且要把目的位置的段地址送 CS。

（2）近转移指令一般用于向下转移，即目的位置的偏移地址大于本转移指令位置的偏移地址，有时也用于向上转移。

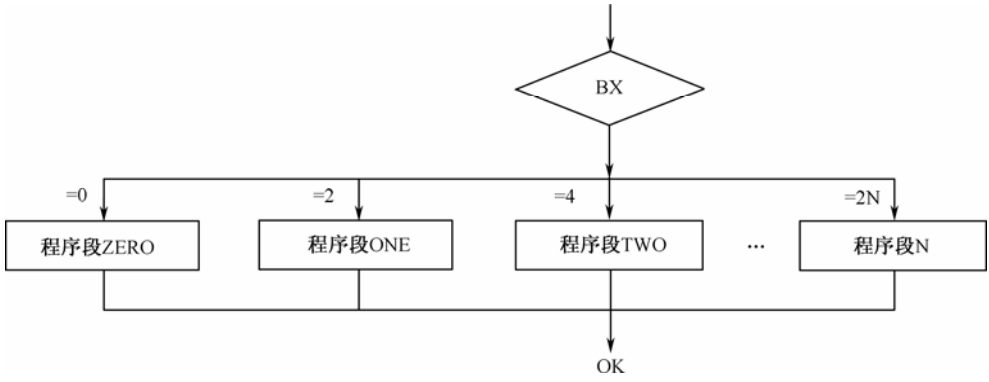


图 6.4　程序流程图

（3）转移指令本身不影响状态标志。

（4）在需要根据两个数的比较结果确定程序流向时，应根据实际问题所涉及的数据是无符号数，还是有符号数来选用对应的条件转移指令。

（5）有些条件转移指令的助记符有其他表示形式，详情见附录 C。

6.3　分支程序设计

分支程序设计首先要在分析实际问题的基础上确定若干个程序分支。再在此基础上，或者选用影响状态标志的指令和条件转移指令，或者选用间接寻址的无条件转移指令来选择转向不同的程序分支。前者称为测试法分支程序设计，后者称为跳转表法分支程序设计。

6.3.1　测试法分支程序设计

测试法分支程序由影响状态标志的指令和条件转移指令实现分支的选择。前一指令用于根据具体要求产生状态标志，后一指令根据状态标志确定转向。常用的影响状态标志的指令有 CMP 等指令。

【例 6.10】编写程序，求 Z=| X − Y |　（X、Y 为有符号数）。

程序一

分析：将有符号数 X 与 Y 相比较，若 X<Y 则将 Y−X 的值送 Z；否则将 X−Y 的值送 Z。由于 X、Y 为有符号数，故使用有符号数专用条件转移指令。程序流程图如图 6.5 所示，程序如下（缺少一条指令请读者补上）。

```
NAME EXAMPLE6_10_1
DSEG   SEGMENT
       X   DB   40H
       Y   DB   73H
       Z   DB   ?
DSEG    ENDS
```

```
;
SSEG   SEGMENT   STACK
       DB   80H   DUP（0）
SSEG   ENDS
;
CSEG   SEGMENT
       ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START: MOV   AX, DSEG
       MOV   DS, AX
       MOV   AL, X
       CMP   AL, Y; 根据 X-Y 产生状态标志
       JL    XL   ; 有符号数 X＜Y 则转 XL
       SUB   AL ,  Y
       MOV   Z ,  AL ; Z ← X-Y
       ;
  XL:  MOV   BL , Y
       SUB   BL, AL
       MOV   Z , BL  ;  Z ← Y-X
       ;
  OK:  MOV   AH, 4CH
       INT   21H     ; 返回 DOS
CSEG   ENDS
       END   START
```

程序二

分析：计算 X−Y，若 X−Y≥0，则直接取其差值；否则将差值取负。减法指令对状态标志的影响与比较指令相同，可根据减法操作对负号标志 SF 的影响确定程序流向。程序流程图如图 6.6 所示，程序如下：

```
NAME EXAMPLE6_10_2
DSEG   SEGMENT
X   DB   40H
Y   DB   73H
Z   DB   ?
DSEG   ENDS
;
SSEG   SEGMENT   STACK
DB   80H   DUP（0）
SSEG   ENDS
;
CSEG   SEGMENT
       ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START:      MOV   AX , DSEG
```

```
        MOV   DS , AX
        MOV   AL , X
        SUB   AL , Y; AL←X−Y, 且产生状态标志
        JNS   XG ; 差值为正数则转 XG 处
        NEG   AL
XG:     MOV   Z , AL
        ;
        MOV   AH, 4CH
        INT   21H; 返回 DOS
CSEG    ENDS
        END   START
```

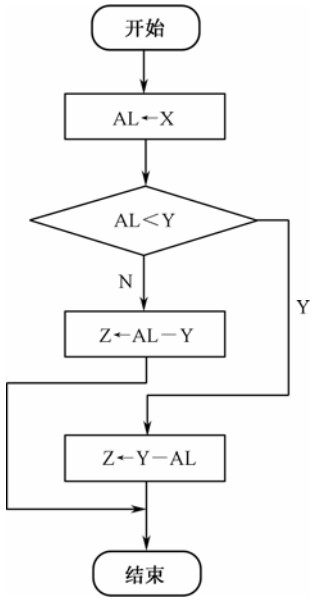


图 6.5　例 6.10 程序一流程图

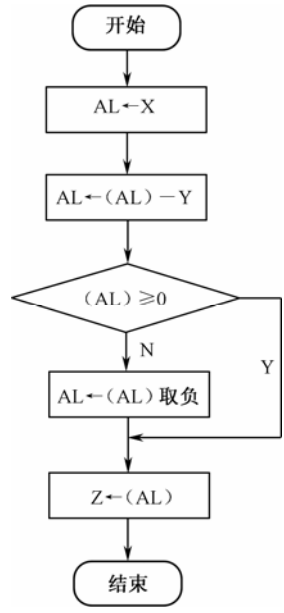


图 6.6　例 6.10 程序二流程图

【例6.11】根据键盘输入的数字 0，1，2 或 3 分别显示信息“8086/8088”，“80386”，“80486”或“Pentium”。

分析：测试键盘输入的数字 AL，若 AL=*i*（*i*=0，1，2）条件成立。则转第 *i* 分支，否则再测试 AL=*i*+1 条件是否成立。程序流程图如图 6.7 所示。程序如下：

```
NAME    EXAMPLE6_11
DSEG    SEGMENT
MESS0   DB      '808/8088', '$'
MESS1   DB      '80386',  '$'
MESS2   DB      '80486', '$'
MESS3   DB      'Pentium', '$'
DSEG    ENDS
;
SSEG    SEGMENT
```

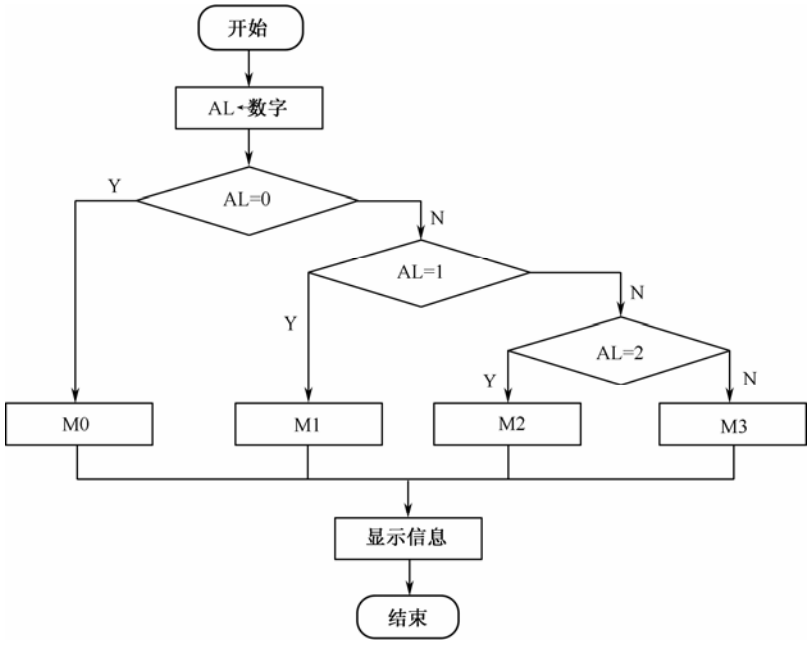


图 6.7　例 6.11 程序流程图

```
            DB    80H   DUP（0）
SSEG    ENDS
;
CSEG     SEGMENT
            ASSUME    DS: DSEG, SS: SSEG, CS: CSEG
START:      MOV    AX, DSEG
            MOV    DS, AX
            ;
            MOV    AH, 8
            INT     21H     ; 从键盘输入一字符, 其 ASCII 码送 AL
            AND    AL, 0FH ; 将 AL 中的 '0', '1', '2' 或 '3'转换为对应的数字 0,1,2 或 3
            CMP     AL, 0
        JZ        M0       ; AL=0 则转 M0 处
        CMP     AL, 1
        JZ        M1       ; AL=1 则转 M1 处
        CMP     AL, 2
        JZ        M2       ; AL=2 则转 M2 处
   M3: MOV    DX, OFFSET    MESS3
        JMP     SHORT DISP
   M0: MOV    DX, OFFSET    MESS0
        JMP     SHORT    DISP
   M1: MOV    DX, OFFSET    MESS1
        JMP     SHORT    DISP
   M2: MOV    DX, OFFSET    MESS2
```

```
        ;
DISP: MOV   AH, 09H
        INT     21H             ; 显示 DX 所指的字符串
        ;
        MOV   AH, 4CH
        INT     21H             ; 返回 DOS
CSEG ENDS
        END   START
```

6.3.2 跳转表法分支程序设计

使用上述测试法固然也可以实现多分支程序设计（如【例 6.11】），但是当分支较多时，程序有三个缺陷：其一，烦琐；其二，多次测试将花费大量时间；其三，到达不同分支的时间不一。使用跳转表法实现多分支程序设计则可避免这些缺陷。

跳转表是一段连续的存储区，根据其内容可以分为分支地址表和转移指令表两种，由此对应两种跳转表法分支程序设计。

1. 分支地址表法

用分支地址表法实现分支程序设计的思想是，通过 DW 指令将 n 个分支入口处的偏移地址如 BRi（i=0,1,…,n−1）按序放在一段连续的存储区中，构成跳转表。设该存储区的首地址为 BASE，第 i 个分支入口处的偏移地址的存放位置则为 BASE+2*i。当程序需要转向第 i 个分支时，只需将 2*i 送 BX，并使用指令

```
JMP   BASE[BX]
```

即可实现。

【例 6.12】编程要求与【例 6.11】同，即根据键盘输入的数字 0，1，2 或 3，分别显示信息“8086/8088”，“80386”，“80486”或“Pentium”。要求配合使用分支地址构成的跳转表实现多分支程序设计。

分析：将显示“8086/8088”，“80386”，“80486”，“Pentium”的多个分支程序入口地址 M0，M1，M2 以及 M3 依次放在基地址为 BASE 的跳转表中，如图 6.8 所示。根据 AL 中的数值计算出对应的分支入口地址相对 BASE 的偏移量，然后用一条无条件转移指令即可转至对应的分支入口。程序如下：

```
NAME     EXAMPLE6_12
DSEG     SEGMENT
BASE     DW   M0   ; 多分支入口处偏移地址构成的跳转表;
         DW   M1
         DW   M2
         DW   M3
MESS0    DB '8086/8088', '$'
MESS1    DB '80386', '$'
MESS2    DB '80486', '$'
MESS3    DB 'Pentium', '$'
DSEG     ENDS
```

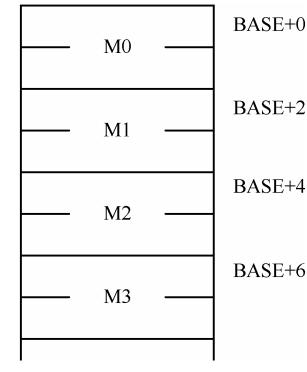


图 6.8　分支地址构成跳转表

```
;
SSEG        SEGMENT
    DB      80H DUP（0）
SSEG        ENDS
;
CSEG        SEGMENT
            ASSUME   DS:DSEG,SS:SSEG,CS:CSEG
START:      MOV   AX , DSEG
            MOV    DS , AX
            ;
            MOV    AH , 8
            INT     21H    ; 从键盘输入 '0', '1', '2' 或 '3', 其 ASCII 码送 AL
            AND    AL , 0FH ; 将输入的 '0', '1', '2', 或 '3' 转换成对应的数字 i
            SHL     AL, 1     ; AL←2*i
            CBW
            MOV    BX, AX ; BX←2*i
            JMP     BASE[BX] ; 利用地址表实现多分支
            …
M0:         MOV   DX, OFFSET   MESS0
            JMP    SHORT   DISP
M1:         MOV   DX, OFFSET   MESS1
            JMP    SHORT   DISP
M2:         MOV   DX, OFFSET   MESS2
            JMP    SHORT   DISP
M3:         MOV   DX, OFFSET   MESS3
;
DISP:       MOV   AH, 09H
            INT   21H                ; 显示 DX 所指的字符串
            ;
            MOV   AH , 4CH
            INT    21H                ; 返回 DOS
CSEG        ENDS
          END   START
```

当要实现远转移，即各个程序分支并不在同一代码段时，则需要使用 DD 伪指令来定义跳转表。

2. *转移指令表法*

转移指令表法与上述的分支地址表法相比有两点区别：其一，构成跳转表的不是 *n* 个分支入口处的偏移地址，而是转向 *n* 个分支的无条件转移指令

JMP　BR*i*

（*i*=0,1,2,……,*n*–1）；其二，跳转表不是用 DW 或 DD 指令定义，而是作为代码段中的一段程序，该跳转表的首址 BASE 不是变量而是标号。当需要转向第 *i* 个分支时，只需将标号 BASE 的偏移地址与 2**i* 之和送 BX，然后使用指令

```
JMP   BX
```

即可使程序转向

```
JMP   BRi
```

指令，进而转向 BRi 分支。

【例 6.13】编程要求与【例 6.12】同，即根据键盘输入的数字 0，1，2 或 3，分别显示信息“8086/8088”，“80386”，“80486”，“Pentium”，要求配合使用转移指令构成的跳转表实现多分支程序设计。

分析：将 4 条无条件短转移指令（该指令占两个字节）依次放在基地址为 BASE 的跳转表中，这些指令的功能分别是转向显示“8086/8088”，“80386”，“80486”及“Pentium”的分支，如图 6.9 所示。与【例 6.12】不同，跳转表不放在数据段，而是放在代码段。程序如下：

```
NAME      EXAMPLE6_13
DSEG      SEGMENT
    MESS0     DB   '8086/8088', '$'
    MESS1     DB   '80386', '$'
    MESS2     DB   '80486', '$'
    MESS3     DB   'Pentium', '$'
DSEG      ENDS
;
SSEG      SEGMENT
          DB    80H DUP（0）
SSEG      ENDS
;
CSEG      SEGMENT
          ASSUME   DS:DSEG,SS:SSEG,CS:CSEG
START:    MOV    AX,DSEG
          MOV    DS , AX
          ;
          MOV    AH,8
          INT    21H       ; 从键盘输入 '0', '1', '2' 或 '3', 其 ASCII 码送 AL;
          AND    AL,0FH    ; 将输入的 '0', '1', '2', 或 '3' 转换成对应的数字 i;
          SHL    AL,1
          CBW              ; AX←2*i;
          MOVBX, OFFSET BASE
          ADD BX, AX; BX←转移目的位置;
          JMP   BX
BASE:     JMP SHORT M0; 转移指令构成的跳转表;
          JMP SHORT M1
```

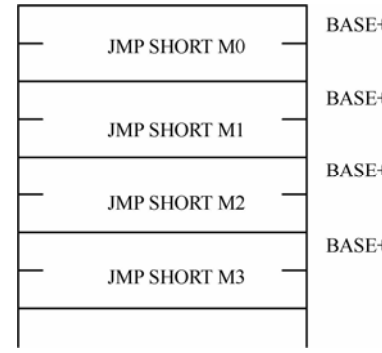


图 6.9　转移指令构成的跳转表

```
        JMP SHORT M2
        JMP SHORT M3
M0:     MOV   DX, OFFSET MESS0
        JMP   DISP
M1:     MOV   DX, OFFSET MESS1
        JMP   DISP
M2:     MOV   DX, OFFSET MESS2
        JMP   DISP
M3:     MOV   DX, OFFSET MESS3
        ;
DISP:   MOV   AH, 09H
        INT   21H  ; 显示 DX 所指的字符串;
        ;
        MOV   AH, 4CH
        INT   21H  ; 返回 DOS。
CSEG    ENDS
        END   START
```

当需要实现超出短转移范围的近转移，或需要实现远转移时，跳转表中的指令就应该相应改为无条件近转移指令或无条件远转移指令，这两种指令分别占 3 字节和 5 字节。对于上例来说，就应该对 BX 内容的设置作相应的修改。具体作何修改，请读者自己思考。

注意：

（1）用测试法进行多分支程序设计时，流程图中对各个条件的测试的先后次序应尽量与所涉及的具体问题提出的先后次序相符；编程中使用多个条件转移指令形成多个程序分支时，其先后次序也应尽量与流程图所示的次序相符。这样处理可以使得编程思路清晰，易读易改。

（2）要根据题意为每个程序分支安排出口，避免某个程序分支错误地顺序进入另一个程序分支。常用方法是在程序分支的结束处使用无条件转移指令使程序转向正确的目的位置。

（3）对于既可以用图 6.1（a）又可以用图 6.1（b）所示的结构编写程序时，宜选用后者。这样处理具有转移次数少，程序结构简单的优点（如【例 6.10】中程序一与程序二所示）。

（4）对于多分支的程序设计，宜选用跳转表法，以使程序具有结构简单，转移次数少，到达各分支所需时间一致的优点（如【例 6.11】,【例 6.12】及【例 6.13】所示）。

（5）在分支程序的调试时，应该通过不同的实验数据来检查每一个程序分支是否正常，从而保证整个分支程序的正确性。

6.4 分支程序设计综合举例

【例 6.14】将 BLKS 为首址的连续 *N* 个字节数传送至以 BLKD 为首址的存储区，编写此数据块传送程序。

分析：

（1）根据题意，两数据块的相对位置有以下 3 种情况：

① 两数据块不重叠，如图 6.10（a）所示。在这种情况下从首部或从尾部开始传送均可以。

② 两数据块有部分重叠，且 BLKS 地址大于 BLKD 地址，如图 6.10（b）所示。在这种情况下，只能从首部开始传送。若从尾部开始传送，则将破坏 BLKS 数据块中尚未传送的首部数据。

③ 两数据块有部分重叠，且 BLKS 地址小于 BLKD 地址，如图 6.10（c）所示。这种情况下，只能从尾部开始传送。

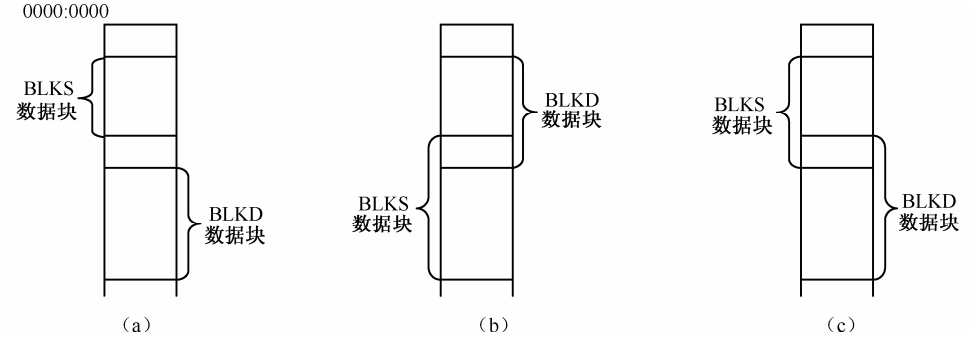


图 6.10　两个数据块相对位置示意图

（2）可以用变址寄存器 SI 指向 BLKS 中的字节数，用 DI 指向 BLKD 中对应的位置，将 SI 所指的字节数送 DI 所指的位置，并使 SI，DI 均指向下一字节。在尚未传送完的情况下转移至完成上述功能的程序段起始位置，直至传送完毕。程序流程图如图 6.11 所示。

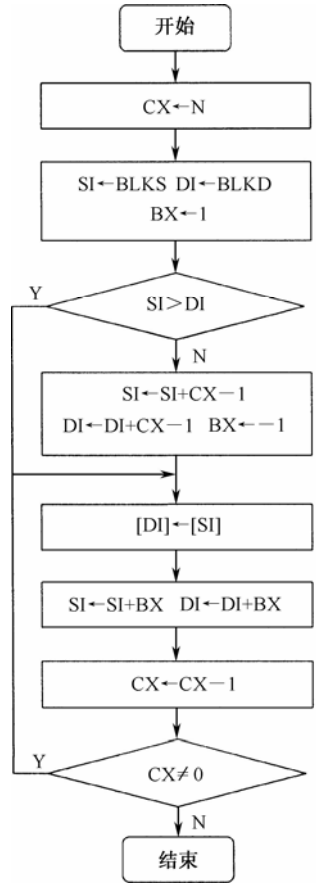


图 6.11　例 6.14 程序流程图

程序如下：

```
NAME EXAMPLE6_14
DSEG        SEGMENT
            ORG     $+24H
   STRG     DB THIS  IS  A  PROGRAM  FOR  STRING
            MOVING'
   N        EQU  $-STRG
   BLKS     DW STRG
   BLKD     DW STRG+5
     DSEG       ENDS
;
SSEG        SEGMENT  STACK
            DB      80H  DUP（0）
SSEG        ENDS
;
CSEG        SEGMENT
            ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
START:      MOV     AX   , DSEG
            MOV     DS   , AX
            MOV     CX   , N            ; CX←数据块字节数;
            MOV     SI   , BLKS         ; SI 指向源数据块首部;
            MOV     DI   , BLKD         ; DI 指向目的数据块首部;
```

```
        MOV     BX   , 1              ; 设置 SI、DI 修正量为 1;
        CMP     SI   , DI
        JA      MOVE                 ; 源数据块首址大于目的数据块首址则转 MOVE 处;
        ADD     SI   , CX
        DEC     SI                   ; SI 指向源数据块尾部;
        ADD     DI , CX              ; DI 指向目的数据块尾部;
        DEC     DI
        NEG     BX                   ; 设置 SI、DI 修正量为－1;
MOVE:   MOV     AL , [SI]
        MOV     [DI] , AL
        ADD     SI   , BX
        ADD     DI   ,BX
        DEC     CX
        JNZ     MOVE; CX≠0, 即尚未传送完毕, 则转 MOVE 处继续传送;
        ;
        MOV     AH, 4CH
        INT     21H ; 返回 DOS
CSEG    ENDS
        END   START
```

【例 6.15】用汇编语言源程序实现以下 C 语言的 switch 语句的功能，语句中变量 a 和 b 为有符号整型（int）变量。

```
switch (a % 8)
{
    case0 :     b = 32 ;
                break ;
    case1 :
    case2 :     b = a + 43 ;
                breack ;
    case3 :     b = 2 * a ;
                breack ;
    case4 :     b -- ;
                breack ;
    case5 :
    case6 :
    case 7:     printf ("function5_6_7") ;
                breack ;
}
```

分析：将多个分支程序入口地址 CASE0、CASE12、CASE12、CASE3、CASE4、CASE567、CASE567 及 CASE567 依次放在基地址为 TAB 的跳转表中。根据 a 除以 8 的余数，计算出对应的分支入口地址相对 BASE 的偏移量，然后用一条无条件转移指令即可转至对应的分支入

口。程序如下：

```
NAME    EXAMPLE6_15
DSEG    SEGMENT
A       DW  ?
B       DW  ?
TAB     DW  CASE0, CASE12, CASE12, CASE3
        DW  CASE4, CASE567, CASE567, CASE567
 MSG    DB  'function 5_6_7$'
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  DS: DSEG, CS: CSEG
START:  MOV   AX  , DSEG
        MOV   DS  , AX
        MOV   AX  , A
        MOV   BX  , AX
        AND   BX  , 7       ; 得到 BX 的低 3 位，实现 a 除 8 的计算
        SHL   BX  , 1
        JMP   TAB[BX]    ; 利用地址表实现多分支
CASE0:  MOV   B, 32D
        JMP   NEXT
CASE12: ADD   AX,  43D
        MOV   B,  AX
        JMP   NEXT
CASE3:  SHL   AX,  1
        MOV   B,  AX
        JMP   NEXT
CASE4:  DEC   B
        JMP   NEXT
CASE567: LEA   DX,  MSG
        MOV   AH,  9
        INT   21H
NEXT:   …
        MOV   AH, 4CH
        INT   21H ; 返回 DOS
CSEG    ENDS
        END  START
```

习　题

6.1　指出下列程序段执行后 AL 内容是什么？

（1）　M

```
    MOV   AL, 60H
    CMP   AL, 0BBH
    JB    L2
L1: MOV   AL, 0BBH
L2: NOP     ; NOP 为空操作
```

（2）　M

```
    MOV   AL, 60H
    CMP   AL, 0BBH
    JL    L2
L1: MOV   AL, 0BBH
L2: NOP
```

（3）　M

```
    XOR   AL, AL
    MOV   BL, 98H
    ADD   BL, 88H
    JNC   L2
L1: INC   AL
L2: NOP
```

（4）　M

```
    XOR   AL, AL
    MOV   BL, 98H
    ADD   BL, 88H
    JNO   L2
L1: INC   AL
L2: NOP
```

（5）　M

```
    MOV   AL, 1
    MOV   BL, 72H
    ADD   BL, 40H
    JC    L2
L1: XOR   AL, AL
L2: NOP
```

（6）　M

```
    MOV   AL, 1
    MOV   BL, 72H
    ADD   BL, 40H
    JO    L2
L1: XOR   AL, AL
L2: NOP
```

（7）　M

```
    MOV   BL, 46H
    TEST BL, 01H
    JZ    L2
L1: XOR   AL, AL
    JMP   OK
L2: MOV AL, 1
OK: NOP
```

（8）　M

```
    MOV   BL, 46H
    TEST  BL, 0FFH
    JNP   L2
L1: XOR   AL, AL
    JMP   OK
L2: MOV AL, 1
OK: NOP
```

6.2　以下程序段用于计算符号函数。

$$Y=\begin{cases}1 & (X>0)\\ 0 & (X=0)\\ -1 & (X<0)\end{cases}$$

X 取值范围为－128～+127。试填写方框中的指令。

M

```
    MOV   AL, X
    CMP   AL, 0
```

```
        JZ      ZERO
        [          ]
        MOV   AL, 1
        [          ]
ZERO: XOR   AL, AL
        [          ]
NEGA: MOV   AL, 0FFH
OK: MOV   Y, AL
```

6.3　指出以下程序执行后 ANS 的内容是什么？

```
DSEG    SEGMENT
D1      DW   9CC0H
D2      DW   1240H
D3      DW   0620H
ANS     DW   ?
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME    DS: DSEG, CS: CSEG
START:  MOV   AX, DSEG
        MOV   DS, AX
        MOV   AX, D1
        CMP   AX, D2
        JG    CMP13
        MOV   AX, D2
        CMP   AX, D3
        JG    OK
D3G:    MOV   AX, D3
        JMP   OK
CMP13:  CMP   AX, D3
        JL    D3G
    OK: MOV   ANS, AX
        MOV   AH, 4CH
        INT   21H
CSEG    ENDS
        ENDS  START
```

6.4　把 3 个连续存放的单字节无符号数按递增次序重新存放在原存放位置。

6.5　有单字节符号数 X 和 Y，求 X^2 及 Y^2 且将 X^2 及 Y^2 中的较大者送变量 Z。要求分别使用本章图 6.1（a），图 6.1（b）所示的两种分支结构编写程序。

6.6　求 5 字节无符号数 D1 和 D2 之和。要求考虑进位问题，结果送 6 字节变量 D3。试编写程序。

6.7　将双字节有符号数 D1 与 D2 之差送变量 D3。要求考虑溢出问题，结果用 4 字节表示。试编写程序。

6.8　根据键盘输入的字符'A'～'E'（或'a'～'e'），分别显示'1st'，'2nd'，'3rd'，'4th' 及 '5th'，当输入的是其他字符则重新输入。试编写程序。

第 7 章　循环程序设计

在解决实际问题时，经常需要重复一些操作，解决这种问题适宜使用循环程序结构。本章首先介绍了两种循环程序结构和实现循环控制的循环指令。在此基础上，对于计数控制，已知最大循环次数的控制，以及循环次数未知的条件控制循环程序的设计方法和技巧进行了详细介绍。还介绍了多重循环程序设计的方法。

7.1　循环程序结构

循环程序一般由 3 个部分组成：

（1）置循环初值部分。这一部分本身仅执行一次，有两方面的作用：其一，为循环工作部分置初值，包括对工作部分所涉及的某些寄存器或存储单元置零或置其他初始值，使地址指针指向一个数据区的起始位置等；其二，为循环控制部分置初值，包括置循环次数或置循环结束条件等。

（2）循环工作部分。这一部分又称为循环体，是需要重复执行的程序段，是循环程序要完成的具体操作。

（3）循环控制部分。这一部分也会随循环工作部分一道重复执行，其作用是修改用于循环计数的寄存器的值，以及对循环结束条件是否成立作出判断。

循环程序结构如图 7.1 所示，其中图 7.1（a）是“先工作后控制”的结构，这种结构下，循环工作部分至少被执行一次；图 7.1（b）是“先控制后工作”的结构，这种结构下允许“0”次循环。

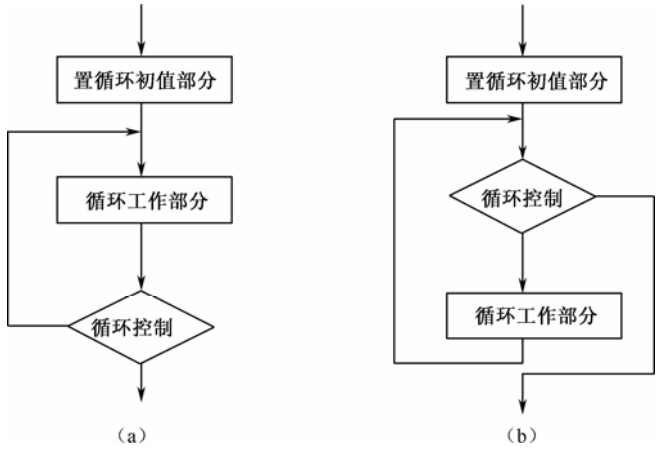


图 7.1　循环程序结构

【例 7.1】上一章【例 6.14】中的程序虽然被用做分支程序结构的示例，但其中一部分可以视为循环程序结构。为便于说明问题，对原要求作以下简化：要求将 BLKS 为首址的连续 *N* 个字节数传送至 BLKD 为首址的存储区，且假设这两个存储区不重叠，则程序流程图如图 7.2 所示。

程序如下：

```
NAME      EXAMPLE7_1
DSEG      SEGMENT
BLKS      DB   （N 个字节数）
N         EQU  $-BLKS
            M
BLKD      DB   N  DUP（?）
DSEG      ENDS
;
SSEG      SEGMENT  STACK
  DB      80H  DUP（0）
SSEG      ENDS
;
CSEG      SEGMENT
        ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
START:    MOV  AX, DSEG
          MOV  DS, AX
          ;
          MOV  CX,  N
          MOV  SI,  OFFSET  BLKS
          MOV DI, OFFSET BLKD;  置循环初
          值部分;
          ;
MOVE:     MOV  AL, [SI]
          MOV  [DI], AL
          INC  SI
          INC  DI;  循环工作部分;
          ;
          DEC  CX
          JNZ  MOVE;  循环控制部分;
          ;
          MOV  AH, 4CH
          INT  21H
CSEG      ENDS
          END START
```

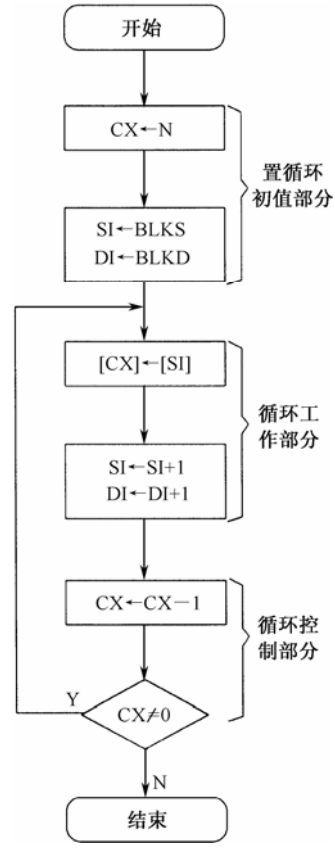


图 7.2　例 7.1 程序流程图

以上程序通过指令

```
DEC  CX
JNZ  MOVE
```

实现循环控制。80x86 指令系统中的一条 LOOP 指令就包括了这两条指令的功能。事实上 80x86 指令系统中设有一组专门的循环指令。

7.2 循环指令

虽然使用前述的转移指令也可以进行循环程序设计，但使用循环指令来进行循环程序设计会更加方便。循环指令分为重复控制指令和串操作指令，其中串操作指令要结合重复前缀才能实现循环功能。

7.2.1 重复控制指令

重复控制指令的一般格式：

XXXX　　短标号

重复控制指令的功能：

将寄存器 CX 默认为重复控制计数器，根据（CX）是否为 0 等情况来控制是转向短标号处还是顺序执行其后的指令，在使用重复控制指令构成循环结构时，指令指定的短标号位置一般在该重复控制指令之上，转向短标号处就实现了对上面程序段的重复执行。

这类指令共有 4 条，如表 7.1 所示。这些指令本身并不影响状态标志。

表 7.1

助 记 符	功　能
LOOP	CX←（CX）−1,（CX）≠0 则转指定的短标号处，否则顺序执行
LOOPZ	CX←（CX）−1,（CX）≠0 且 ZF=1 则转指定的短标号处，否则顺序执行
LOOPNZ	CX←（CX）−1,（CX）≠0 且 ZF=0 则转指定的短标号处，否则顺序执行
JCXZ	（CX）=0 则转指定的短标号处，否则顺序执行

1. LOOP指令

该指令首先使寄存器 CX 中的计数值减 1，然后判断（CX）是否为零。若（CX）≠0 则转至该指令指定的短标号处，即重复执行短标号开始的程序段；否则顺序执行该 LOOP 指令后的指令，即不再重复执行短标号开始的程序段。

【例 7.2】以下程序段的功能是，求 BUFF 数据区中各字节之和，并送 SUM 变量。

```
          M
BUFF      DB   40H, 82H, 0F2H, 05H, 0, 10H, 0, 18H
SUM       DW   ?
          M
          XOR   AX, AX   ; 用做累加器的 AX 置初值 0;
          MOV   SI, OFFSET BUFF   ; 用做数据区指针的 SI 指向数据区起始位置;
          MOV   CX, 8     ; 用做重复控制计数器的 CX 置循环次数 8;
          ;
AGAIN:    ADD   AL, [SI]
          ADC   AH, 0
          INC   SI        ; 循环工作部分;
          LOOP  AGAIN   ; 循环控制。
          MOV   SUM, AX
          M
```

程序中指令

```
LOOP   AGAIN
```

实现重复控制。（CX）减 1 后非零（循环体的执行次数尚未达到 8）则重复累加工作；否则不再重复，而执行其后的指令

```
MOV   SUM, AX
```

2. LOOPZ指令

该指令与 LOOP 指令的区别仅在于，在判断（CX）是否为零的同时还要判断 ZF 标志。若（CX）≠0 且 ZF=1 则转至该指令指定的短标号处；否则顺序执行 LOOPZ 指令后的指令。

3. LOOPNZ指令

该指令与 LOOP 指令的区别仅在于，若（CX）≠0 且 ZF=0 则转至该指令指定的短标号处；否则顺序执行 LOOPNZ 指令后的指令。

【例 7.3】以下程序的功能是，求 BUFF 数据区中第一个零元素之前的各字节数之和，并送 SUM 变量。

```
          M
BUFF      DB   40H, 82H, 0F2H, 05H, 0, 10H, 0, 18H
SUM       DW   ?
          M
          XOR   AX, AX
          MOV   SI, OFFSET BUFF
          MOV   CX, 8
          ;
AGAIN:    ADD   AL, [SI]
          ADC   AH, 0
          INC   SI
          ;
          CMP BYTE PTR[SI], 0      ; 将下一个字节数与零做比较, 比较结果将影响 ZF 标志;
          LOOPNZ AGAIN             ; 循环控制。
          MOV SUM, AX
          M
```

程序中的指令

```
LOOPNZ AGAIN
```

实现重复控制。（CX）减 1 后非零且 CMP 指令使 ZF=0，即 BUFF 数据区中各字节数尚未累加完毕，且下一个字节数非零，则重复累加工作；否则不再重复，而执行其后的指令。

4. JCXZ指令

该指令与 LOOP 指令的区别有两点：其一是没有使 CX 中计数值减 1 的功能；其二是当（CX）=0 时转至该指令指定的短标号处，否则顺序执行该 JCXZ 指令后的指令。

7.2.2 串操作指令及重复前缀

串（STRING）是指存储器中的字节序列或字序列。串操作指令对字节串或字串进行每次一个元素的操作。在串操作指令前加上重复前缀，就可由硬件重复执行该串操作指令，使处理串的操作速度远远大于使用重复控制指令处理的速度。串操作指令通常不带操作数，重复前缀只能配合串操作指令使用。两者如表 7.2 所示。

表 7.2 串操作指令及重复前缀

	助 记 符	功 能	备 注
串操作指令	MOVSB MOVSW	ES：[DI]←DS：[SI]，DI←（DI）±1，SI←（SI）±1 ES：[DI]←DS：[SI]，DI←（DI）±2，SI←（SI）±2	若方向标志 DF=0 则“+1”；否则“−1”。若方向标志 DF=0 则“+2”否则“−2”
	LODSB LODSW	AL←DS：[SI]，SI←（SI）±1 AX←DS：[SI]，SI←（SI）±2	
	STOSB STOSW	ES：[DI]←（AL），DI←（DI）±1 ES：[DI]←（AX），DI←（DI）±2	
	CMPSB CMPSW	根据 DS：[SI]−ES：[DI]产生状态标志，DI←（DI）±1，SI←（SI）±1 根据 DS：[SI]−ES：[DI]产生状态标志，DI←（DI）±2，SI←（SI）±2	
	SCASB SCASW	根据（AL）−ES：[DI]产生状态标志，DI←（DI）±1 根据（AX）−ES：[DI]产生状态标志，DI←（DI）±2	
重复前缀	REP	CX←（CX）−1，（CX）≠0 则重复所缀串指令	
	REPZ	CX←（CX）−1，（CX）≠0 且 ZF=1 则重复所缀串指令	
	REPNZ	CX←（CX）−1，（CX）≠0 且 ZF=0 则重复所缀串指令	

1. MOVSB、MOVSW指令（串传送指令）

将 DS: SI 所指的源操作数传送到 ES: DI 所指的目的位置，指令 MOVSB 实现字节传送，MOVSW 实现字传送；然后修改 SI、DI 的内容使之指向下一元素位置。对 SI、DI 的修改取决于两个因素：其一是操作数的属性；其二是方向标志 DF 的状态。具体修改方法如下：

（1）MOVSB 指令，若 DF=0 则 SI←（SI）+1，DI←（DI）+1；否则 SI←（SI）−1，DI←（DI）−1。

（2）MOVSW 指令，若 DF=0 则 SI←（SI）+2，DI←（DI）+2；否则 SI←（SI）−2，DI←（DI）−2。

该指令不影响状态标志。在其前加上重复前缀 REP 可实现串从一个存储区到另一个存储区的传送。在使用重复前缀的情况下，应预先将重复次数送寄存器 CX，并根据需要使用指令 CLD 使 DF 置“0”，或使用指令 STD 使 DF 置“1”。

【例 7.4】以下程序的功能等同于【例 7.1】中的程序功能。

```
NAME   EXAMPLE7_4
DSEG   SEGMENT
BLKS   DB （N 个字节数）
N      EQU $-BLKS
         M
BLKD    DB N DUP（?）
```

```
DSEG    ENDS
;
SSEG    SEGMENT   STACK
        DB      80H DUP（0）
SSEG    ENDS
;
CODE    SEGMENT
        ASSUME   DS: DSEG, ES: DSEG, SS: SSEG, CS: CSEG
START: MOV   AX, DSEG
       MOV   DS, AX; 置源串段地址;
       MOV   ES, AX; 置目的串段地址;
       ;
       MOV   CX, N; 置串操作重复次数;
       MOV   SI, OFFSET   BLKS; 置源串偏移地址;
       MOV   DI, OFFSET   BLKD; 置目的串偏移地址;
       CLD; DF←0, 以使其后每次传送工作之后地址指针作“+”修改;
       ;
       REP   MOVSB; 重复 N 次字节传送操作;
       ;
       MOV   AH, 4CH
       INT   21H; 返回 DOS。
CSEG   ENDS
       END   START
```

2. LODSB、LODSW指令（串装入指令）

与 MOVSB、MOVSW 指令的区别仅在于，这两条指令所作数据传送的目的位置分别为 AL、AX，不涉及寄存器 DI。

3. STOSB、STOSW指令（串存储指令）

与 MOVSB、MOVSW 指令的区别仅在于，这两条指令源操作数分别为（AL）、（AX），不涉及寄存器 SI。在其前加上重复前缀 REP 可以使一个存储区各单元置同一数据。

【例 7.5】以下程序的功能是使 BUFF 数据区各单元存放用户输入的同一个字符。

```
NAME       EXAMPLE 7_5
DSEG       SEGMENT
BUFF       DB   100 DUP（?）
DSEG       ENDS
;
SSEG       SEGMENT   STACK
           DB   80H DUP（0）
SSEG       ENDS
;
```

```
CSEG        SEGMENT
            ASSUME   ES: DSEG, SS: SSEG, CS: CSEG
START:      MOV   AX, DSEG
            MOV   ES, AX
            ;
            MOV   AH, 01H
            INT   21H; 等待用户输入字符, 且将该字符送 AL;
            ;
            MOV   DI, OFFSET   BUFF; DI 指向目的串初始位置;
            MOV   CX, 100; 置串操作重复次数;
            CLD; DF←0, 使其后的串操作指令对地址指针作“+”修改;
            ;
          REP   STOSB; 将（AL）送 BUFF 数据区各字节单元, 重复前缀 REP 的功能是使所缀指
                       令 STOSB 重复 CX 初值所指定的次数。
            ;
            MOV   AH, 4CH
            INT   21H; 返回 DOS
CSEG        ENDS
            ENDS   START
```

4. CMPSB、CMPSW指令（串比较指令）

将 DS: SI 所指的源操作数减去 ES: DI 所指的目的操作数，即 DS: [SI]－ES: [DI]（注意：不是 ES: [DI]－DS: [SI]），但不回送差值，只是根据减法运算产生状态标志。这两条指令对 SI、DI 的修改分别与 MOVSB、MOVSW 相同。

5. SCASB、SCASW 指令（串搜索指令）

与 CMPSB、CMPSW 指令的区别仅在于，这两条指令所规定的被减数分别为（AL）、（AX），不涉及寄存器 SI。在其前加上重复前缀 REPZ 或 REPNZ，可以在 ES: DI 所指的一个存储区中寻找与（AL）、（AX）不等或相等的元素。

6. REP前缀

使所缀的串操作指令重复 CX 初值所指定的次数。在使用该前缀及其串操作指令前，应根据需要对 CX 设置初值。REP 前缀常配合 MOVSB、MOVSW、STOSB、STOSW 指令使用。

例如，在【例 7.5】程序中，REP 前缀所缀的串操作指令 STOSB 的功能只是将（AL）送 ES：DI 所指的字节单元，且对 DI 的内容作“+1”修改，STOSB 指令本身并无重复功能。而 REP STOSB 则使得 STOSB 指令重复执行，重复次数取决于此前设置的 CX 的初值。

7. REPZ前缀

当所缀的串操作指令尚未执行 CX 初值所指定的次数，且所缀串操作指令的执行使得 ZF=1，则重复执行所缀的串操作指令；否则顺序执行串操作指令的下一指令。REPZ 前缀常配合 CMPSB、CMPSW、SCASB、SCASW 指令使用。

8. REPNZ前缀

与 REPZ 前缀的区别仅在于，使所缀的串操作指令重复执行的条件不同。当所缀的串操作指令尚未执行 CX 初值所指定的次数，且所缀串操作指令的执行使得 ZF=0，则重复执行所缀的串操作指令；否则顺序执行串操作指令的下一指令。

【例 7.6】以下程序的功能是，将 BUFF 数据区中紧接第一个零元素后的字节数送 ANS 单元（设在末元素前存在零元素）。

```
DSEG    SEGMENT
BUFF    DB   40H, 82H, 0F2H, 05H, 0, 10H, 0, 18H
ANS     DB   ?
DSEG    ENDS
;
SSEG    SEGMENT
        DB   80H DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
        ASSUME   ES: DSEG, SS: SSEG: CS: CSEG
START:  MOV   AX, DSEG
        MOV   ES, AX
        XOR   AL, AL; 置搜索对象—0;
        MOV   DI, OFFSET   BUFF; DI 指向搜索区起始位置;
        MOV   CX, 8; 置最大搜索次数;
        CLD        ; DF←0, 使其后的串操作指令对地址指针作“+”修改;
        REPNZ   SCASB; 根据 0-ES: [DI]影响标志 ZF, 且 DI←(DI)+1, CX←(CX)-1, 若(CX)
                       ≠0 且 ZF=0(未重复指定次数且未找到零元素)则重复执行 SCASB
                       指令, 否则执行下一指令;
        MOV   AL, ES: [DI]
        MOV   ANS, AL; 将紧接第一个零元素后的字节数送 ANS 单元;
        MOV   AH, 4CH
        ;
        INT   21H; 返回 DOS。
CSEG    ENDS
        ENDS   START
```

说明：

（1）在 80386 以上指令集中，串不仅可以是字节序列和字序列，而且可以是双字序列。前述对应双字序列的串操作指令为

```
MOVSD
LODSD
STOSD
```

CMPSD

SCASD

这类指令所涉及的源串指针为 ESI，目的串指针为 EDI，寄存器为 EAX，指针修正值为 4。例如

CLD

SCASD; 根据（EAX）−ES: [EDI]产生状态标志, EDI←（EDI）+4。

（2）在 80386 以上指令集中，还有一类与输入、输出相关的串指令。串输入指令：

INSB

INSW

INSD

这 3 条指令的功能分别是将（DX）端口的 1 个字节、1 个字或 1 个双字输入到 ES: DI（或 ES: EDI）所指的目的位置，且修改 DI（或 EDI）的内容，使之指向下一目的位置。修改的方法与前述串指令相同。

串输出指令：

OUTSB

OUTSW

OUTSD

这 3 条指令的功能分别是将 DS: SI（或 DS: ESI）所指的 1 个字节、1 个字或 1 个双字输出到（DX）端口，且修改 SI（或 ESI）的内容，使之指向下一源位置。修改的方法与前述串指令相同。串输入和串输出指令均不影响状态标志。

（3）在 80386 以上指令集中，重复控制指令和重复前缀所涉及的重复控制计数器默认为 ECX，即重复次数可达 2^{32}。

（4）有些重复控制指令、串操作指令和重复前缀还有其他的表示形式，详见附录 C。

7.3 循环程序设计

7.3.1 计数控制的循环程序设计

在循环程序设计中，用计数的方法实现循环控制是一种基本且常用的方法。这种方法适用于已知循环次数的场合。在使用这种方法进行循环程序设计时，应根据题意确定需要重复进行的操作，将其作为循环工作部分；在循环工作部分之前对循环工作部分所涉及的某些寄存器、存储单元置初始值，对计数寄存器 CX（或 ECX）置循环次数；还应根据计数寄存器的计数情况控制循环，通常使用 LOOP 指令实现，在循环部分只是一条串操作指令时，则可通过在该串操作指令前加上 REP 前缀来实现。

【例 7.7】编写计算 $N!$ 的程序。

分析：

（1）循环工作部分包括乘法运算。考虑到对于一个不大的 N 值，$N!$ 的数值也会很大，为提高程序的适应性，故使用 32 位寄存器，如使用 EAX 作为累乘器。在使用乘法指令时，EAX 既提供被乘数，又存放乘积。使用另一个 32 位寄存器提供乘数 i（i=1,2,⋯N），考虑到 LOOP 指令的功能，故宜使用 ECX 提供乘数。

（2）置循环初值部分包括：作为累乘器的 EAX 应置初值 1。现在来考虑为 ECX 置什么

初值。若置初值 1，则在循环工作部分应使其增 1，且在循环控制部分要判断（ECX）是否等于 N 值。考虑到 LOOP 指令本身具有使 ECX 减 1，且根据（ECX）是否为 0 来确定是否重复执行循环工作部分的功能，故宜对 ECX 置初值 N。

（3）循环控制只要使用 LOOP 指令即可。程序流程图如图 7.3 所示。程序如下：

```
NAME      EXAMPLE7_7
.386（或 .486, .586）; 选择 80386（或 80486、Pentium）指令集;
DSEG      SEGMENT
N         EQU（一个自然数）
ANS       DD  ?
DSEG      ENDS
;
SSEG      SEGMENT  TACK
          DB  80H DUP（0）
SSEG      ENDS
;
CSEG      SEGMENT  USE16; 16 位段;
          ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
START:    MOV  AX, DSEG
          MOV  DS, AX
          ;
          MOV  ECX, N
          MOV  EAX, 1; 置循环初值;
NEXT:     IMUL  EAX, ECX; 循环工作;
          LOOP  NEXT; 循环控制;
          MOV  ANS, EAX
          ;
          MOV  AH, 4CH
          INT  21H; 返回 DOS。
CSEG      ENDS
          END  START
```

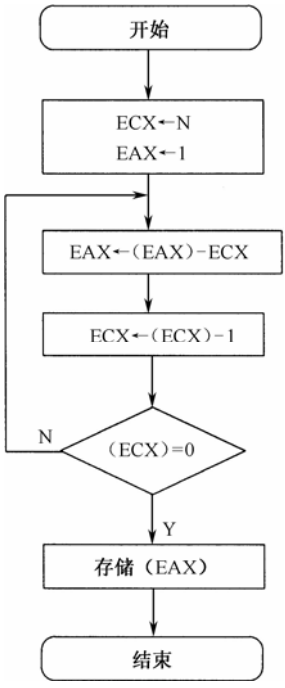


图 7.3　例 7.7 程序流程图

编写此程序时，假定 $N!$ 值不超过 32 位寄存器的表示范围。

【例 7.8】编写一程序，以统计 BUF 字数据区中负数的个数。

分析：

（1）循环工作部分包括：将 BUF 数据区中的一个数与 0 相比较，若该数小于 0 则使存放负数个数的寄存器，如 AL 增 1；否则不作此增 1 工作。为了使每一次操作针对 BUF 数据区中的一个数据，需要使用一个通用寄存器，如 BX 作地址指针。于是需要重复进行的操作包括对地址指针的修改。

（2）置循环初值部分包括：作为存放负数个数的寄存器 AL 应置初值 0，地址指针 BX 应指向 BUF 数据区起始位置，计数寄存器 CX 应置 BUF 数据区中数据的个数。

（3）循环控制只要使用 LOOP 指令即可。程序流程图如图 7.4 所示。程序如下：

```
NAME      EXAMPLE7_8
DSEG      SEGMENT
BUF       DW   0, 8200H, 42H, 0FFFFH, 1200H, 3203H
          DW   0C000H, 9030H, 6800H, 10H, 08H, 2222H
COUNT     EQU（$-BUF）/2; 使 COUNT 等于 BUF 数据区中
          数据个数;
ANS       DB   ?
DSEG      ENDS
;
SSEG      SEGMENT   STACK
          DB   80H DUP（0）
SSEG      ENDS
;
CSEG      SEGMENT
          ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START:    MOV   AX, DSEG
          MOV   DS, AX
          ;
          XOR   AL, AL
          MOV   BX, OFFSET BUF
          MOV   CX, COUNT; 置循环初值;
          ;
NEXT:     CMP   WORD PTR[BX], 0
          JGE   GEZ; BX 所指的有符号双字节数不为负数
          则转标号 GEZ 处;
          INC   AL
GEZ:      INC   BX
          INC   BX; 循环工作;
          LOOP   NEXT; 循环控制;
          MOV   ANS, AL
          ;
          MOV   AH, 4CH
          INT   21H; 返回 DOS。
CSEG      ENDS
          END   START
```

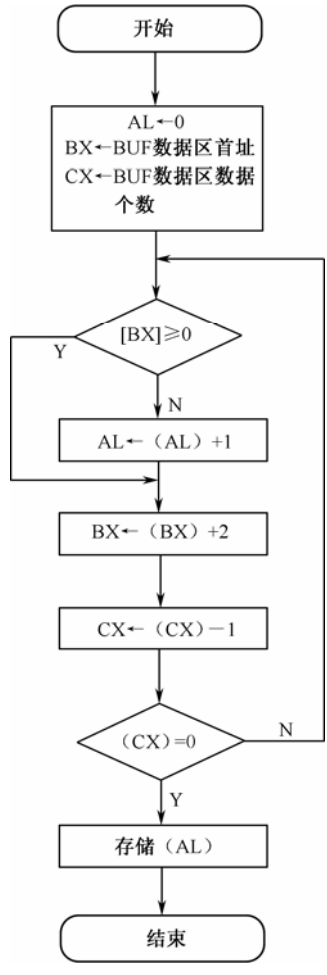


图 7.4　例 7.8 程序流程图

7.3.2　条件控制的循环程序设计

1. 已知最大循环次数的条件控制循环程序设计

在使用循环结构解决问题中，有时只是已知最大循环次数。循环工作部分的重复次数是

否应实际达到最大循环次数，需要根据循环工作部分的执行情况来确定。解决这类问题通常有以下 3 种方法。

（1）使用计数控制的循环程序设计方法，并在循环工作部分根据执行情况判断是否中途跳出循环。

【例 7.9】编写一程序，用以判断 BUF1 和 BUF2 两个等长度的数据区中数据是否相同。相同则使 FLAG 单元置 0，否则置−1。

分析：

① 循环工作部分包括使地址指针作“+1”修改，将 BUF1 数据区中的一个数据与 BUF2 数据区中同序号的数据作比较。

② 置循环初值部分包括对存放比较结果的 FLAG 单元置初值 0，这一点用伪指令即可实现。地址指针应预先指向数据区起始位置的前一字节。对 CX 置最大循环次数。

③ 循环控制一方面使用 LOOP 指令，使循环工作部分的执行次数不超过 CX 的初始值；另一方面在循环工作部分增加一分支结构，当同序号的两个数据不同则使 FLAG 置−1，且跳出循环，否则转 LOOP 指令。程序流程图如图 7.5 所示。程序如下：

```
NAME    EXAMPLE7_9
DSEG    SEGMENT
BUF1    DB    （N 个字节数）
BUF2    DB    （N 个字节数）
COUNT   EQU   $－BUF2
FLAG    DB    0
DSEG    ENDS
;
SSEG        SEGMENT   STACK
            DB   80H   DUP（0）
SSEG        ENDS
;
CSEG        SEGMENT
            ASSUME   DS: DSEG, SS: SSEG,
                     CS: CSEG
START:      MOV   AX, DSEG
            MOV   DS, AX
            ;
            MOV   SI, OFFSET   BUF1−1
            MOV   DI, OFFSET   BUF2−1
            MOV   CX, COUNT; 置循环初值;
            ;
NEXT:       INC   SI
            INC   DI
            MOV   AL, ［SI］
            CMP   AL, ［DI］
```

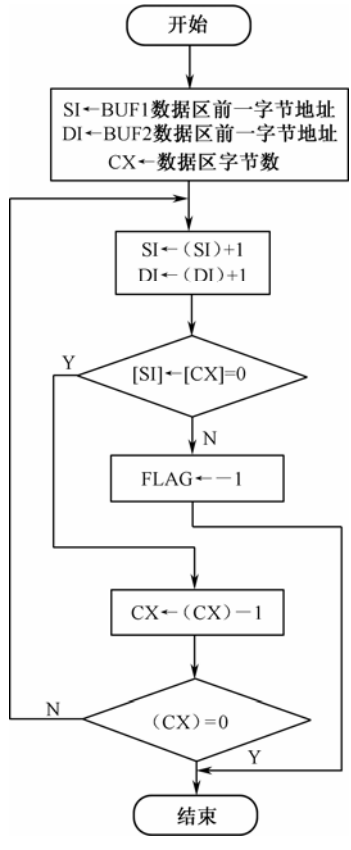


图 7.5　例 7.9 程序流程图

```
                JZ    LOOP1
                MOV   FLAG, –1
                JMP   OK; 循环工作, 其中亦含循环控制;
LOOP1:          LOOP  NEXT; 循环控制;
                ;
OK:             MOV   AH, 4CH
                INT   21H; 返回 DOS。
CSEG            ENDS
                END   START
```

（2）使用 LOOPZ、LOOPNZ 指令实现循环的条件控制。

【例 7.10】要求实现的程序功能与【例 7.9】同。

分析：循环工作部分及置循环初值部分与【例 7.9】类似。循环工作部分是否再次执行既取决于循环工作部分的已执行次数，又取决于两个同序号的数据的比较结果。具体控制为：当循环工作部分的已执行次数未达到 CX 的初始值，且刚作比较的两个同序号的数据相同，则再次执行循环工作部分；否则结束循环。考虑到 LOOPZ 指令的功能，故使用该指令实现循环控制。

值得注意的是，采用这种循环控制方法时，循环的结束或是因为循环工作部分的执行次数已达到 CX 的初始值，或是因为已发现两个数据区中两个同序号的数据不同。于是，可以接着根据 ZF 的值予以判断，并根据判断结果确定是否将“-1”送 FLAG 单元。程序流程图如图 7.6 所示。程序如下：

```
NAME      EXAMPLE7_10
DSEG      SEGMENT
BUF1      DB   （N 个字节数）
BUF2      DB   （N 个字节数）
COUNT     EQU  $–BUF2
FLAG      DB   0
DSEG      ENDS
;
SSEG      SEGMENT   STACK
          DB   80H   DUP（0）
SSEG      ENDS
;
CSEG      SEGMENT
          ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
START:    MOV   AX, DSEG
          MOV   DS, AX
          MOV   SI, OFFSET   BUF1–1
          MOV   DI, OFFSET   BUF2–1
          MOV CX, COUNT; 置循环初值;
          ;
```

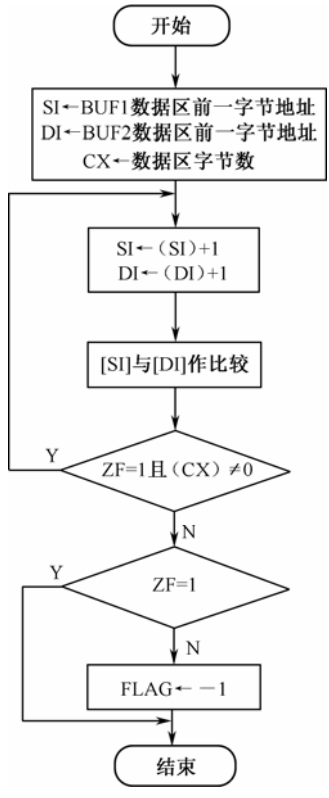


图 7.6　例 7.10 程序流程图

```
NEXT:   INC   SI
        INC   DI
        MOV   AL, [SI]
        CMP   AL, [DI]; 循环工作;
        LOOPZ   NEXT; 循环控制;
        JZ   OK
        MOV   FLAG, -1
    ;
  OK:   MOV   AH, 4CH
        INT   21H; 返回 DOS。
CSEG    ENDS
        END   START
```

若在此程序中将指令

```
MOV   SI, OFFSET   BUF1-1
MOV   DI, OFFSET   BUF2-1
```

改为

```
MOV   SI, OFFSET   BUF1
MOV   DI, OFFSET   BUF2
```

且将指令

```
INC   SI
INC   DI
```

改放到 CMP 指令之后，这样修改是否正确，请读者思考。

（3）使用带有 REPZ、REPNZ 前缀的串比较、串搜索指令实现循环的条件控制。

【例 7.11】要求实现的程序功能与【例 7.9】同。

分析：考虑到 REPZ 前缀结合 CMPSB 指令可以实现两个存储器操作数的比较，实现变址寄存器 SI、DI 的增 1（或减 1），实现计数寄存器 CX 减 1，并根据比较结果及（CX）是否为 0 来控制是否进行后续数据的比较。因此，可将其用于本例题的循环程序设计。值得注意的是，CMPSB 指令所涉及的两个操作数分别用 DS: [SI]和 ES: [DI]给出，故应该使段寄存器 DS 和 ES 均存放 DSEG 数据段的段地址；若在循环工作部分之前使 SI 指向 BUF1 数据区首址，使 DI 指向 BUF2 数据区首址，则应同时使用 CLD 指令，使 DF 置“0”，以便 CMPSB 指令使 SI、DI 作增 1 修改。程序如下：

```
NAME    EXAMPLE7_11
            M
CSEG    SEGMENT
        ASSUME   DS: DSEG, ES: DSEG, SS: SSEG, CS: CSEG
START:  MOV   AX, DSEG
        MOV   DS, AX
        MOV   ES, AX
        ;
        MOV   SI, OFFSET BUF1
```

```
                MOV    DI, OFFSET BUF2
                MOV    CX, COUNT
                CLD; 置循环初值
                REPZ   CMPSB; 循环工作及循环控制
                JZ     OK
                MOV    FLAG, -1
                ;
OK:             MOV    AH, 4CH
                INT    21H; 返回 DOS
CSEG            ENDS
                END    START
```

2. 循环次数未知的条件控制循环程序设计

在循环程序设计中，有时对循环次数无法预知，对此可根据循环工作部分的执行所产生的条件来确定是否继续执行循环工作部分。

【例 7.12】编写一程序，求满足 $\sum_{i=1}^{x} i < 8\,000$ 的最大的 X 值。

分析：

（1）循环工作部分包括：将一个作为累加器的寄存器，如 AX 加上 i 值，i 值可使用另一寄存器，如 BX 提供。在相加前，先对 BX 作增 1 修改。

（2）置循环初值部分包括：对作为累加器的 AX 置初值 0，对 BX 置初值 0。

（3）对于 AX 被加上（BX）后的内容作判断，若（AX）<7 000 则继续执行循环工作部分，否则结束循环。结束循环时的（BX）是首次使 $\sum_{i=1}^{x} i < 8\,000$ 不成立的值，故将（BX）减去 1 所得到的值就是所求的答案。

程序流程图如图 7.7 所示。

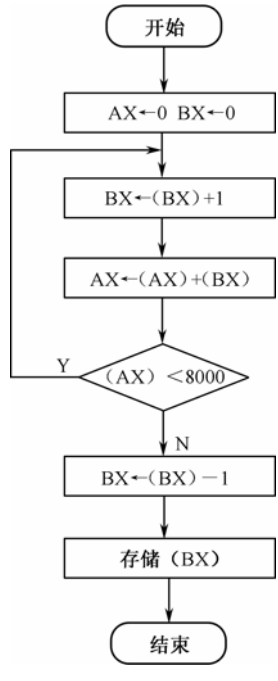


图 7.7　例 7.12 程序流程图

程序如下：

```
NAME    EXAMPLE7_12
DSEG    SEGMENT
CONS    EQU 8000
X       DW  ?
DSEG    ENDS
;
SSEG    SEGMENT  STACK
        DB  80H DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
START:  MOV  AX, DSEG
```

```
        MOV   DS, AX
        ;
        XOR   AX, AX
        XOR   BX, BX; 置循环初值;
NEXT:   INC   BX
        ADD   AX, BX; 循环工作;
        ;
        CMP   AX, CONS
        JB    NEXT ; 循环控制;
        DEC   BX
        MOV   X, BX
        ;
        MOV   AH, 4CH
        INT   21H; 返回 DOS。
CSEG    ENDS
        END   START
```

不难看出，循环次数未知的条件控制循环程序设计可以使用测试法分支程序设计方法，即使用影响状态标志的指令及条件转移指令来实现循环的控制。

7.3.3 多重循环程序设计

前面讲述的循环程序设计涉及的均是单循环程序。单循环程序的特点是，其循环工作部分或是顺序程序段，或是分支程序段。当一个循环程序的循环工作部分又包含循环程序时，就构成了多重循环程序。以多重循环中的二重循环为例，其程序结构如图 7.8 所示。若该图所示的内层循环工作部分还包含循环结构，则循环就多于二重。

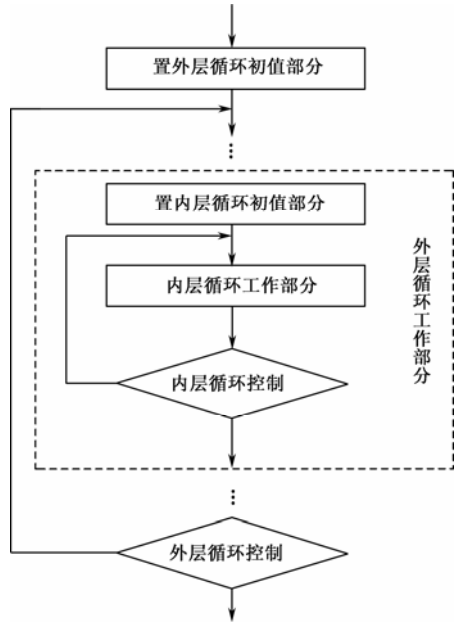


图 7.8　二重循环程序结构

【例 7.13】编写一程序，用以统计 BUF 数据区的 64 个字节中“0”二进制位的数目。

分析：

（1）一个字节中“0”二进制位数目的统计可以用重复次数为 8 的循环结构解决；对 64 个字节的处理可以用重复次数为 64 的循环结构解决；将前一循环结构作为后一循环结构的循环工作部分，即使用二重循环就可以完成题目给定的功能。

（2）内层循环的构成。

- 循环工作部分包括：判断一个字节，如 SI 所指字节的某一位是否为“0”，为“0”则使计“0”寄存器，如 BX 增 1；否则跳过 BX 增 1 操作。
- 置循环初值部分包括：计“0”寄存器 BX 置初值 0，循环计数寄存器 DH 置初值 8。
- 循环控制部分包括：对循环计数寄存器 DH 作减 1 计数，若减 1 后（DH）≠0，则重复执行循环工作部分，否则结束循环。

（3）外层循环的构成。

- 循环工作部分包括：内层循环，修改地址指针 SI 以及将内层循环产生的（BX），即一个字节中“0”二进制位的数目累加到某个内存单元，如 COUNT 单元。
- 置循环初值部分包括：作为存放所有字节中“0”二进制位数目的内存单元 COUNT 置初值 0，这一工作可通过伪指令实现；地址指针 SI 指向 BUF 数据区起始位置；外循环计数器 CX 置初值 64。
- 循环控制使用 LOOP 指令即可。

程序流程图如图 7.9 所示。程序如下：

```
NAME    EXAMPLE7_13
DSEG    SEGMENT
BUF     DB（64 个字节数）
COUNT   DW 0
DSEG    ENDS
;
SSEG    SEGMENT  STACK
        DB  80H DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
START:  MOV  AX, DSEG
        MOV  DS, AX
        ;
        MOV  SI, OFFSET BUF
```

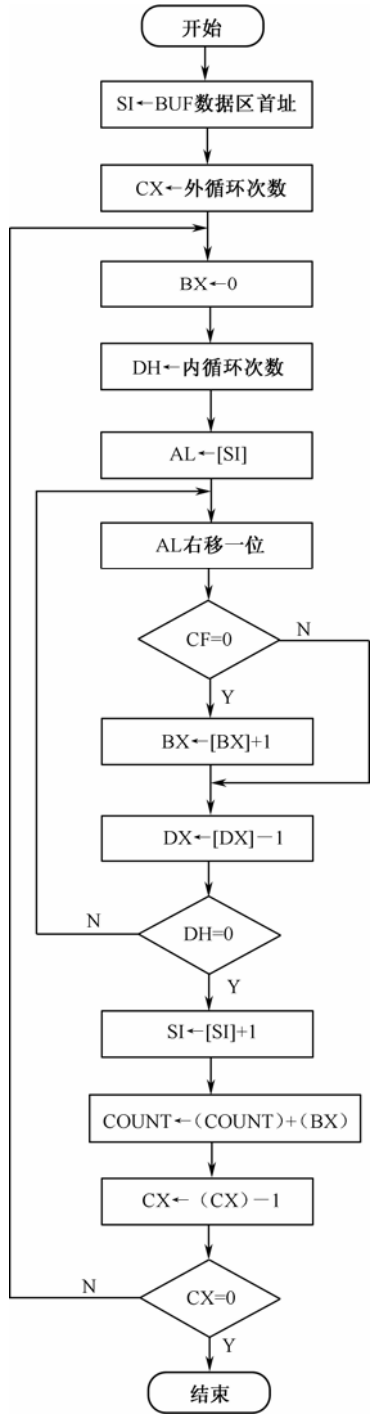


图 7.9　例 7.13 程序流程图

```
        MOV   CX, 64; 置外层循环初值;
        ;
EXL:    XOR   BX, BX
        MOV   DH, 8
        MOV   AL, [SI]; 置内层循环初值;
        ;
INL:    ROR   AL, 1
        JC    NEXT
        INC   BX; 内层循环工作;
        ;
NEXT:   DEC   DH
        JNZ   INL; 内层循环控制;
        INC   SI
        ADD   COUNT, BX; 外层循环工作;
        LOOP  EXL; 外循环控制;
        ;
        MOV   AH, 4CH
        INT   21H; 返回 DOS。
CSEG    ENDS
        END   START
```

对该程序请读者思考两个问题：其一，内层循环为何不使用 LOOP 指令作循环控制；其二，在该程序中 BX 用做一个字节的计“0”寄存器，每当一个字节中“0”二进制位的个数统计完毕，则将（BX）加到 COUNT 单元中，这一相加工作要进行 64 次。如果将 BX 改作所有字节的计“0”寄存器，最后一次性将（BX）送 COUNT 单元，则程序将更为简洁，而且程序执行将更快，为此程序应如何修改？

注意：

（1）使用 LOOPZ、LOOPNZ 指令实现循环控制时，控制的一个依据是减 1 后的 CX（或 ECX）值，另一个依据是前一条影响 ZF 标志的指令产生的 ZF 标志。为了提供后一个依据，就应根据题意在循环工作部分设置合适的影响 ZF 标志的指令，而且要避免该指令所产生的 ZF 标志被其他指令所破坏。例如，【例 7.10】程序后所述的修改使得 CMP 指令所产生的状态标志被随后的两条 INC 指令破坏，故修改后的程序就不能满足题意。

（2）在进行循环程序设计时，要根据题意区分循环工作部分和置循环初值部分各自应作的操作。以【例 7.7】为例，若将属于置循环初值的操作

```
MOV   EAX, 1
```

放到了循环工作部分，即把标号 NEXT 改放到该指令前，则实际求得的就不是 $N!$，而是 1。若将属于置循环初值的操作

```
MOV   ECX, N
```

放到了循环工作部分，即把标号 NEXT 改放到该指令前，则将构成死循环。

（3）尽可能使用带有重复前缀的串操作指令实现循环程序设计。这样的循环程序较为简洁，而且运行速度较快。如【例 7.10】和【例 7.11】中两个程序相比，宜选用后者。

7.4 循环程序设计综合举例

【例 7.14】编写一程序，以实现 N 个无序的有符号字节数组由大到小的排序。

分析：设该数组各元素为 X_1，$X_2 \cdots X_n$，可以使用以下排序方法：

（1）首先将 X_1 与 X_2 比较，若 $X_1 \geqslant X_2$，则不做交换，否则交换 X_1 与 X_2，然后将 X_2（可能已是交换后的值）与 X_3 比较，按同样原则确定是否交换。依次类推，直到对 X_{n-1} 与 X_n 做同样工作。当这第一轮工作结束时，X_n 成为数组元素中的最小者。

（2）对数组元素 X_1，$X_2 \cdots X_{n-1}$ 做同样工作，当第二轮工作结束时，X_{n-1} 成为数组各元素中的次小者。

（3）经过 $n-1$ 轮工作，就可以实现数组由大到小的排序。

（4）每一轮工作可作为内层循环，而 $n-1$ 次内层循环构成外层循环。事实上，有时并不需要 $n-1$ 次内层循环即可完成数组的排序工作。为了避免不必要的循环，可以设置一个交换标志。如 BL 在内层循环工作部分开始前置初值“0”。在内层循环工作部分若发生两个元素的交换，则对其置“-1”。内层循环结束后检查交换标志，若为“0”则排序完成，否则进行下一次循环。

程序流程图如图 7.10 所示，程序如下：

```
NAME      EXAMPLE7_14
DSEG      SEGMENT
X         DB   0, 34H, 90H, 0FFH, 81H, 11H,
          07H, 10H
COUNT     EQU $-X
DSEG      ENDS
;
CSEG      SEGMENT
          ASSUME   DS: DSEG, CS: CSEG
START:    MOV   AX, DSEG
          MOV   DS, AX
          MOV   DX, COUNT-1;  置外层循
环初值;
          ;
EXL:      XOR   BL, BL
          MOV   CX, DX
```

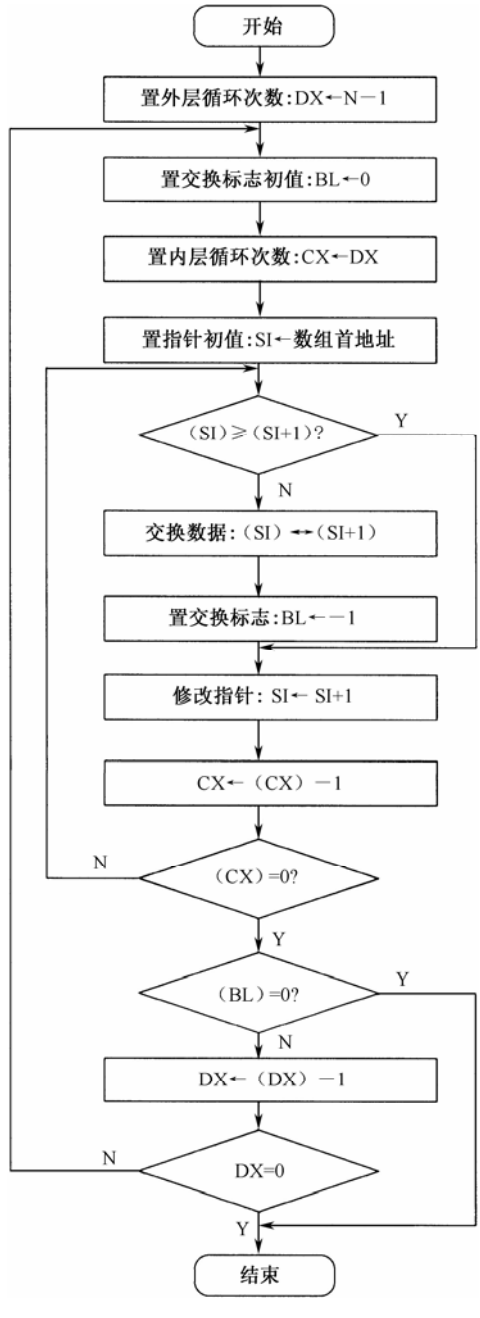


图 7.10　例 7.14 程序流程图

```
        MOV   SI, OFFSET X; 置内层循环初值;
        ;
INL:    MOV   AL, [SI]
        CMP   AL, [SI+1]
        JGE   NEXT
        ;
        XCHG  AL, [SI+1]
        MOV   [SI], AL
        MOV   BL, 0FFH
NEXT:   INC   SI; 内层循环工作;
        LOOP  INL; 内层循环控制;
        CMP   BL, 0
        JZ    OK; 外层循环工作;
        ;
        DEC   DX
        JNZ   EXL; 外层循环控制;
        ;
OK:     MOV   AH, 4CH
        INT   21H; 返回 DOS。
CSEG    ENDS
        END   START
```

本例内层循环采用计数控制方法，而外层循环采用已知最大循环次数的条件控制方法。

【例 7.15】编写一程序，以实现以下矩阵相乘。

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

分析：

（1）计算公式为：$c_i = \sum_{j=1}^{4} a_{ij} b_j \quad (i = 1,2,3)$

（2）这一问题宜用双重循环处理。内层循环实现单个 c_i 的计算，循环次数为 4；外层循环使内层循环重复 3 次，以实现 $c_i (i = 1,2,3)$ 的计算。

（3）内层和外层循环控制均采用 LOOP 指令，即用 CX 做循环计数。为避免内层循环计数对外层计数的影响，在内层循环之初和之后，分别使用了“PUSH　CX”指令和“POP　CX”指令。

不妨设

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} = \begin{bmatrix} 3 & 0 & 2 & 3 \\ 1 & 1 & 2 & 0 \\ 1 & 0 & 2 & 3 \end{bmatrix}$$

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

程序流程图如图 7.11 所示，程序如下。

```
NAME    EXAMPLE7_15
DSEG    SEGMENT
A       DB   3, 0, 2, 3
        DB   1, 1, 2, 0
        DB   1, 0, 2, 3
B       DB   1, 0, 0, 1
C       DW   3 DUP (? )
DSEG    ENDS
;
CSEG    SEGMENT
        ASSUME   DS: DSEG, CS: CSEG
START:  MOV   AX, DSEG
        MOV   DS, AX
        MOV   SI, 0
        MOV   BX, 0
        MOV   CX, 3
LOOP1:  PUSH   CX
        MOV   DI, 0
        MOV   WORD PTR C[BX], 0
        MOV   CX, 4
LOOP2:  MOV   AH, 0
        MOV   AL, A[SI]
        MUL   B[DI]
        ADD   C[BX], AX
        INC   SI
        INC   DI
        LOOP   LOOP2
        ADD   BX, 2
        POP   CX
        LOOP   LOOP1
        ;
        MOV   AH, 4CH
        INT   21H;  返回 DOS。
CSEG    ENDS
        END   START
```

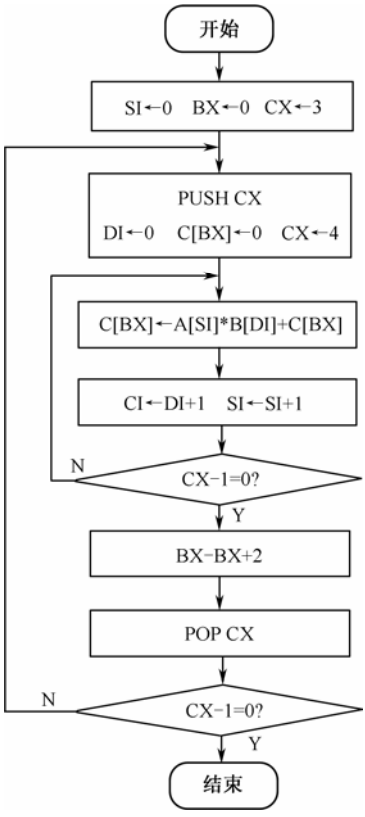


图 7.11　例 7.15 程序流程图

习　题

7.1　试述循环程序结构与分支程序结构之间的关系。

7.2　说明编程者在循环程序设计中通常将 CX 作为循环计数器的原因。

7.3　根据给定程序段填空

（1）
```
BUFF   DB   08H，09H，03H，04H，0，01H
          M
        XOR   BX，BX
        XOR   AX，AX
        MOV   CX，6
NEXT：ADD   AL，BUFF[BX]
        AAA
        INC    BX
        LOOP   NEXT
          M
```
循环结束时　（AX）=_____。

（2）
```
BUFF   DB 0，0，0，96H，23H，0，21H，0
          M
        XOR   AL，AL
        LEA   SI，BUFF-1
        MOV   CX，8
NEXT：INC    SI
        CMP   AL，[SI]
        LOOPZ   NEXT
        MOV   AL，[SI]
          M
```
最后一条指令执行后（AL）=_____。

（3）
```
BUFF   DW   0，3240H，4000H，45H，0C32H，0FFFFH，0，01H
   ORD   DW ?
   M
   MOV   AX，0C32H
   MOV   DI，OFFSET BUFF
   MOV   CX，8
   CLD
   REPNZ   SCASW
   MOV   ORD，DI
```
最后一条指令执行后 ORD 单元内容为_____。

7.4　将习题 7.3（2）的程序用带有重复前缀的串操作指令来改写，要求写出完整程序。

7.5　根据给定功能填写方框中的指令（或伪指令）。

（1）将 STRG 为首址的连续 *N* 个双字传送到 STRG+5 为首址的存储区。

```
DSEG    SEGMENT
STRG    DD （N 个双字）
N       EQU （$−STRG）/4
DSEG    ENDS
;
SSEG    SEGMENT STACK
        DB  80H  DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT USE16
        ASSUME  DS: DSEG, ES: DSEG, SS: SSEG, CS: CSEG
START:  MOV  AX, DSEG
        MOV  DS, AX
        MOV  CX, N
        MOV  SI, OFFSET STRG+4*N−4
        MOV  DI, OFFSET STRG+5+4*N−4
        STD
        MOV  AH, 4CH
        INT  21H
CSEG    ENDS
        END  START
```

（2）将 BUF 数据区中零、正数及负数的个数分别送 ZERO、POSI 及 NEGA 存储单元。

```
DSEG    SEGMENT
BUF     DB  0, 0FFH, 82H, 44H, 91H, 0, 10H, 0
N       EQU  $−BUF
ZERO    DB 0
POSI    DB 0
NEGA    DB 0
DSEG    ENDS
SSEG    SEGMENT  STACK
        DB 80H DUP（0）
SSEG    ENDS
CSEG    SEGMENT
        ASSUME DS: DSEG, SS: SSEG, CS: CSEG
START:  MOV  AX, DSEG
        MOV  DS, AX
```

```
        MOV   SI, OFFSET BUF-1
        MOV   CX, N
NEXT:   [          ]
        MOV   AL, [SI]
        CMP   AL, 0
        JZ    ZE
        JL    NE
        INC   POSI
        [          ]
ZE:     INC   ZERO
        [          ]
NE:     INC   NEGA
LLL:    LOOP  NEXT
        MOV   AH, 4CH
        INT   21H
CSEG    ENDS
        END START
```

7.6　统计 BUF 字数据区第一个零数据后的各数据中正数及负数的个数，并分别送 POSI 及 NEGA 存储单元，试编写程序。

7.7　将 BUF 有符号双字数据区中的最大数送 MAX 单元，最小数送 MIN 单元。

7.8　在 STR 字符串中删除指定字符，要删除的字符通过键盘输入。要求删除一个字符后，后续字符相应前移。试编写程序。

7.9　设数据区 G0，G1，…，G9 分别放有 10 位学生的成绩，每个学生有 5 门课程成绩。现要求统计每个学生 5 门课程成绩的平均值，并放在第 5 门课程成绩之后。试编写程序。

第 8 章　子程序设计及系统调用

本章在介绍子程序基本概念，介绍子程序的定义、调用及返回的基本方法的基础上，着重讲解子程序调用与返回过程中现场的保护和参数的传递方法。还介绍了嵌套子程序和递归子程序的概念和编程要点。对于 MS-DOS 提供的常用内部子程序及其调用方法也进行了介绍，以便读者通过系统功能调用方便地对计算机系统作功能扩充开发。

8.1　调用程序与子程序

循环程序设计技术能够解决在同一程序中连续重复执行同一个程序段的问题，但是对于非连续地重复执行同一程序段的问题则无能为力。对于在不同的代码段，以及在不同的源程序中用到同一个程序段的问题也不能用循环程序设计技术来解决。对于这些场合，为了避免编制程序中的重复劳动，节省程序代码所占的存储空间，往往将需要重复或经常使用的程序段编制成独立的程序，在需要的位置使用特定的指令调用该独立的程序，执行后再返回到上述的调用位置继续执行其后的指令。这里所说的独立的程序就称为子程序，也称为过程，而调用子程序的程序称为调用程序。

8.2　调用与返回指令

实现上述的子程序调用以及子程序返回需要分别使用调用与返回指令。就子程序是否与调用程序在同一代码段而言，调用分为近调用和远调用；就指令中是否直接给出指示子程序的标号而言，调用分为直接调用和间接调用。如表 8.1 所示。

表 8.1　调用指令与返回指令

指　令	功　能
CALL 近标号	下一指令偏移地址入栈，转到近标号指定的段内位置，即实现直接近调用
CALL MEM16/REG16	下一指令偏移地址入栈，转到 16 位存储单元/寄存器指定的段内位置，即实现间接近调用
CALL FAR PTR 远标号	下一指令段地址、偏移地址依次入栈，转到远标号指定的段外位置，即实现直接远调用
CALL　MEM32	下一指令段地址、偏移地址依次入栈，转 32 位存储单元指定的段外位置，即实现间接远调用
CALL　REG32	下一指令段地址、偏移地址依次入栈，转到 32 位寄存器指定的段外位置，即实现间接远调用。此为 80386 以上指令集指令
RET	前一 CALL 指令执行时入栈的地址出栈，使程序转到前一 CALL 指令的下一指令位置
RET　n	在 RET 指令功能的基础上，使堆栈指针下移 n（正偶数）个字节

从前面章节有关内容可知，指令指针 IP（或 EIP）总是存放要执行的下一指令的偏移地址，而 CS 总是存放要执行的下一指令的段地址。大多数指令都遵循这样的规律：在取指令和执行指令后，总是自动使指令指针加上该指令的字节数，使之指向顺序排列的下一指令。而 CALL 指令与转移、循环等指令一样，并不遵循这一规律。该指令将顺序排列的下一指令的地址压入堆栈以便返回，而将该指令给出的转移目标地址送 IP（或 EIP），也即将该指令指

定位置的指令作为要执行的下一指令。如果该 CALL 指令为远调用指令，CS 的内容将是该指令指定位置的段地址。RET 指令也不遵循上述规律，该指令将处于栈顶的地址弹至 IP（或 EIP）以实现返回。对于远调用而言，RET 指令还要恢复调用前 CS 的内容。

【例 8.1】设在执行以下程序段之前（BX）＝4200H，（SP）＝1034H，则指令指针 IP，堆栈指针 SP 及堆栈在 CALL 指令和 RET 指令执行前后的状况如图 8.1 所示。

程序段（在其左边给出了各条指令的存放地址）：

```
                M
D020: 0083      CALL   BX
D020: 0085      NOP
          M
D020: 4200      …
          M
D020: 42B1      RET
```

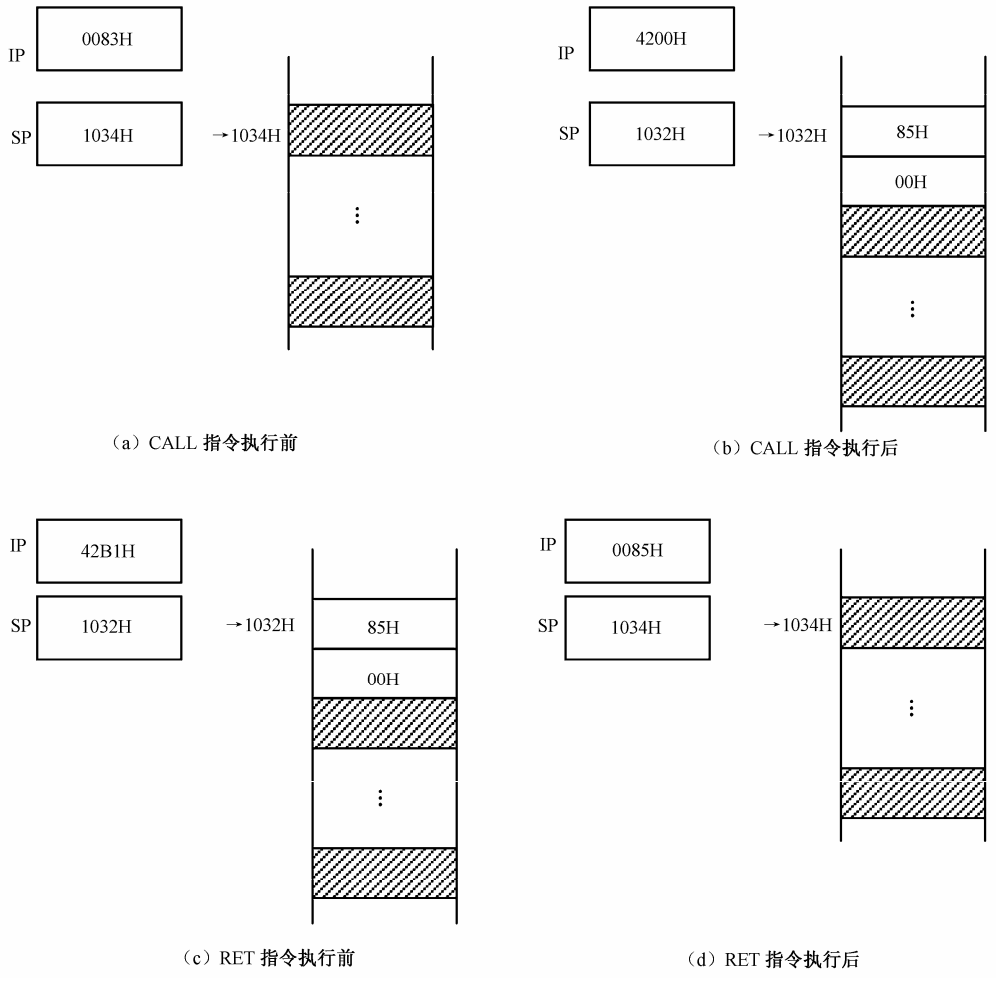


图 8.1　示意图

8.3 子程序设计

进行子程序设计要做以下工作：

（1）将需要重复或经常使用的程序段编制成独立程序，并通过一定的格式来定义，该工作称为子程序的定义。

（2）在调用程序的若干个适当位置使用 CALL 指令调用子程序。

（3）在子程序中设置 RET 指令，以便返回调用程序。

（4）在子程序起始处保护调用时的现场，在子程序返回前恢复调用时的现场。所谓现场，是指有关寄存器及存储单元的内容。保护和恢复现场的目的在于，避免因调用子程序而破坏调用程序所使用的寄存器和内存单元内容。

（5）实现调用程序与子程序之间的参数传递。包括调用程序为子程序提供入口参数，子程序向调用程序提供出口参数。

8.3.1 子程序的定义

1. 子程序定义的格式

子程序按照过程形式定义，其格式有以下两种：

（1）标号 PROC NEAR；其中 NEAR 可省略。
M
标号 ENDP

（2）标号 PROC FAR
M
标号 ENDP

其中格式（1）定义的子程序可供近调用，即段内调用；格式（2）定义的子程序不仅可供近调用，还可供远调用，即段间调用。伪指令 PROC 与 ENDP 相当于一对括号，将子程序实现的工作包含在括号中。作为一对括号，PROC 和 ENDP 前的标号必须一致，如：

```
SUB1  PROC
      M
SUB1  ENDP
```

2. 子程序的结构

子程序在遵循上述格式的基础上， 通常有如下结构：

```
标号  PROC      NEAR 或 FAR
      保护现场
      根据入口参数进行处理
      产生出口参数
      恢复现场
      RET
标号  ENDP
```

8.3.2 子程序的调用与返回

下面通过例子来讲述子程序的调用与返回。

【例 8.2】以下程序用于计算 $C_m^n = m!/(n!*(m-n)!)$ 的值（m，n 为自然数，且 $m>n$）。

分析：

（1）解决这一问题需要三次计算阶乘值 $X!$，而且并非连续重复这一计算工作，故宜于将计算阶乘值 $X!$的工作用子程序来实现。在调用程序中于三个适当位置使用 CALL 指令来调用子程序。在子程序中设置 RET 指令返回调用程序。

（2）不妨使用【例 7.7】所示的方法计算阶乘值。于是，在调用子程序计算 $m!$，$n!$及$(m-n)!$时，应分别将 m，n 及 $m-n$ 的值送 ECX，即为子程序提供入口参数。

（3）子程序根据入口参数计算出对应的阶乘值，并将其作为出口参数通过 EAX 提供给调用程序。

（4）该子程序使用 ECX 传递入口参数，用 EAX 传递出口参数，子程序的执行也影响状态标志，但与调用程序无关。除此以外没有用到其他寄存器和存储单元，故无须保护现场和恢复现场的工作。

调用程序和子程序的程序流程图如图 8.2 所示。程序如下:

```
NAME      EXAMPLE8_2
.386（或: .486、.586）; 选择 80386（或 80486、Pentium）指令集
DSEG      SEGMENT
M         EQU   （一个自然数）
N         EQU   （一个自然数）
ANS       DD    ?
DSEG      ENDS
;
SSEG      SEGMENT   STACK
          DB   80H   DUP（0）
SSEG      ENDS
;
CSEG   SEGMENT   USE16; 16 位段;
          ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START:    MOV   AX, DSEG
          MOV   DS, AX
          MOV   ECX, N
          CALL   SUB1;  调用子程序,  计算 n!;
          MOV   EBX, EAX; EBX←n!;
          MOV   ECX, M
          CALL   SUB1;  调用子程序,  计算 m!;
          DIV    EBX; EAX←m!/n!
          MOV   EBX, EAX
          MOV   ECX, M
```

```
                SUB    ECX, N
                CALL   SUB1; 调用子程序, 计数（m−n）!;
                XCHG   EBX, EAX
                DIV    EBX; EAX←m!/n!/（m−n）!。
                MOV    ANS, EAX
                MOV    AH, 4CH
                INT    21H
;
SUB1        PROC
                MOV    EAX, 1
NEXT:       MUL    EAX, ECX
                LOOP   NEXT
                RET
```

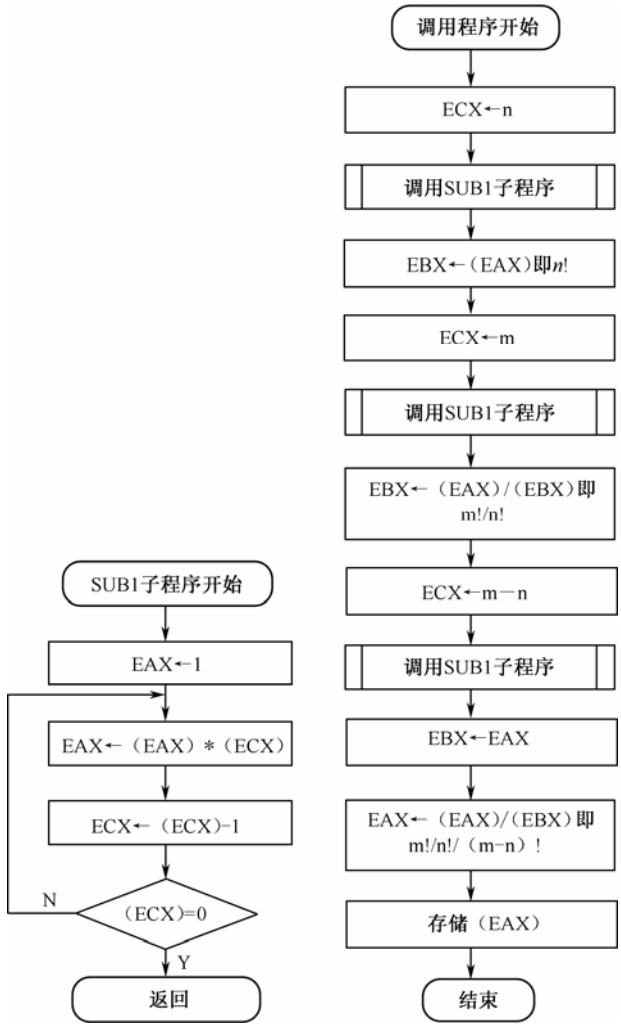


图 8.2　例 8.2 程序流程图

```
SUB1    ENDP
;
CSEG    ENDS
;
        END     START
```

以上程序中，调用程序和子程序处于同一个代码段，CALL 指令通过标号给出转子的目标位置 SUB1，采用的是近直接调用。例如，在执行第一条 CALL 指令时，将当前的 IP 内容，即将其下一指令

```
MOV  EBX，EAX
```

的偏移地址压入堆栈，然后将标号 SUB1 的偏移地址送 IP，从而使程序转向 SUB1 子程序。当子程序执行到 RET 指令时，将先前压入堆栈的 CALL 指令下一指令的偏移地址弹至 IP，从而使程序返回到调用指令的下一指令。

【例 8.3】以下程序功能与【例 8.2】程序相同，但调用程序与子程序不处于同一代码段，采用的是远直接调用。

```
CSEG    SEGMENT; CSEG 代码段开始
        ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
          M
        CALL    SUB1
        MOV     EBX, EAX
          M
        CALL    SUB1
          M
        CALL    SUB1
        XCHG    EBX, EAX
        DIV     EBX
        MOV     ANS, EAX
        MOV     AH, 4CH
        INT     21H
CSEG    ENDS; CSEG 代码段结束;
            M
SUBC    SEGMENT; SUSC 代码段开始;
        ASSUME      CS: SUBC
SUB1    PROC    FAR
          M
        RET
SUB1    ENDP
SUBC    ENDS    ; SUBC 代码段结束。
        END   START
```

程序采用的是远调用，故调用时既要将当前的 IP 内容，又要将当前的 CS 内容入栈，而子程序返回时将入栈的内容弹回这两个寄存器。例如，在执行第一条 CALL 指令时，将当前

的 CS 内容，即将 CSEG 段地址压入椎栈，将当前 IP 内容，即将其下一指令

MOV　EBX，EAX

的偏移地址压入堆栈，然后将标号 SUB1 所在段的段值，即 SUBC 段的段地址送 CS，将标号 SUB1 的偏移地址送 IP，从而使程序转向 SUB1 子程序。当子程序执行到 RET 指令时，将栈顶内容分别弹至 IP 和 CS，从而使程序返回到调用指令的下一指令。

【例 8.4】以下程序是近间接调用的示例。

```
DSEG   SEGMENT
TABW   DW   SUB0, SUB1, SUB2, …SUB9
             M
DSEG       ENDS
;
CSEG       SEGMENT
           ASSUME   DS: DSEG, CS: CSEG
MAIN:      MOV   AX, DSEG
           MOV   DS, AX
           MOV   BX, OFFSET   TABW
           ;
           MOV   AH, 1
           INT   21H; 等待输入一数字字符（′0′～′9′）并送 AL;
           ;
           XOR   AH, AH
           AND   AL, 0FH; 将 AL 中的数字字符‘i’转换成对应数字 i 并送 AX;
           ;
           ADD   AX, AX
           ADD   BX, AX; BX 指向第 i 个子程序入口;
           CALL   WORD   PTR[BX]; 近间接调用。
            M
       MOV   AH, 4CH
       INT   21H
;
SUB0   PROC
        M
SUB0   ENDP
;
SUB1   PROC
        M
SUB1   ENDP
;
        M
SUB9   PROC
        M
```

```
SUB9   ENDP
;
CSEG  ENDS
       END   MAIN
```

程序根据输入的数字来确定调用 10 个子程序之一。指令

```
CALL   WORD   PTR[BX]
```

不是直接给出指示子程序的标号，而是通过 WORD PTR[BX]间接给出转去的目标位置。这样就可以在使用同一条调用指令的情况下，根据 BX 的具体内容转向不同的子程序。

8.3.3 保护现场与恢复现场

1. 保护现场与恢复现场的必要性

在子程序调用前后，子程序中所用到的寄存器或存储单元内容可能发生变化，即子程序调用前的现场被破坏。在子程序返回调用程序时，调用程序将在被破坏的现场下执行其后的程序段，从而难以达到预期的目的。于是，有必要在子程序调用前后分别进行保护现场与恢复现场的工作。

2. 保护现场与恢复现场的位置

通常使用入栈指令实现现场的保护，而使用出栈指令实现现场的恢复。保护现场与恢复现场可以在调用程序中，也可以在子程序中进行。

（1）在调用程序中保护现场与恢复现场。具体而言，就是在 CALL 指令前保护现场，而在 CALL 指令后恢复现场。

【例 8.5】设在所调用的子程序 SUB1 中有可能改变寄存器 BX、CX 及标志寄存器 FLAGS 的内容，而调用程序要求在子程序返回后，能够在 BX、CX 及 FLAGS 原有内容的基础上继续执行其后的程序段，可以使用以下方法：

```
       M
PUSH   BX
PUSH   CX
PUSHF
CALL   SUB1
POPF
POP    CX
POP    BX
       M
```

当然将各个寄存器内容入栈的次序应当与将其出栈的次序相反，这是由堆栈的特点所决定的。

（2）在子程序中保护现场与恢复现场。具体而言，就是在子程序起始处保护现场，而在其返回指令，即 RET 指令前恢复现场。这种方法较为常用。

【例 8.6】以下方法也能达到【例 8.5】所提出的要求。

```
SUB1  PROC
       PUSH   BX
```

```
        PUSH   CX
        PUSHF
          M
        POPF
        POP    CX
        POP    BX
        RET
SUB1    ENDP
```

3. 保护现场与恢复现场的简便方法

在 80286、80386 及其以上指令集中具有将所有通用寄存器入栈及出栈的专用指令。如需要在子程序 SUB1 中保护与恢复 AX、CX、DX、BX、SP、BP、SI、DI 及 FLAGS 的内容，则可以用以下方法：

```
SUB1    PROC
        PUSHA
        PUSHF
          M
        POPF
        POPA
        RET
SUB1    ENDP
```

又如，需要在子程序 SUB1 中保护与恢复 EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI 及 EFLAGS 的内容，则可以用以下方法：

```
SUB1    PROC
        PUSHAD
        PUSHFD
          M
        POPFD
        POPAD
        RET
SUB1    ENDP
```

8.3.4 参数的传递

调用程序在调用子程序之前，往往要将需要子程序处理的原始数据提供给子程序，即为子程序提供入口参数。子程序根据入口参数进行一系列处理后，往往要将处理结果，即出口参数提供给调用程序。这种调用程序为子程序提供入口参数，子程序将出口参数提供给调用程序的工作称为参数的传递。

调用程序与子程序之间传递参数的方式必须事先约定。在设计子程序前，必须确定其入口参数从何处取，处理结果往何处送。一旦按照事先的约定设计子程序，则无论哪个调用程序，无论在调用程序的哪个位置调用该子程序，均必须在 CALL 指令前将入口参数送到约定之处，在 CALL 指令后从约定之处取得出口参数。

常用的参数传递方法有约定寄存器法、约定存储单元法及堆栈法。

1. 约定寄存器法

约定寄存器法是指事先约定使用某些寄存器进行入口参数、出口参数的传递。

【例 8.7】编写程序，用以统计字节数组中零元素的个数。

分析：

（1）为了可以对存放于不同位置、字节数各异的不同的字节数组实现题目提出的功能，可以将统计字节数组中零元素个数的工作用子程序来实现。

（2）对于一个首地址为 ARRAYB，字节数为 COUNT 的字节数组来说，应将 ARRAYB 及 COUNT 作为入口参数提供给子程序；而子程序应在统计出 ARRAYB 开始的 COUNT 个字节数中零元素个数后，将零元素个数作为出口参数提供给调用程序。调用程序最后将出口参数送存储单元，如 ANS 单元。

（3）在编写程序和子程序前，约定使用寄存器来实现入口参数和出口参数的传递。考虑到子程序需要对 ARRAYB 开始的各个字节数判断其是否为零元素，将 ARRAYB 作为 SI 的初值，且逐次使 SI 增“1”，则可用[SI]表示各个字节数。于是可以约定，使用寄存器 SI 传递入口参数 ARRAYB。考虑到子程序需要重复判断[SI]是否为零元素，重复次数为 COUNT，而可实现控制重复的 LOOP 指令使用寄存器 CX 实现重复计数，于是可以约定，使用寄存器 CX 传递入口参数 COUNT。

可以在子程序中用一个寄存器，如 AX 统计数组中零元素的个数，并约定使用该寄存器将零元素的个数作为出口参数传递给调用程序。

调用程序和子程序之间的参数传递可表示为：

SI←ARRAYB（入口参数）

CX←COUNT（入口参数）

AX←零元素个数（出口参数）

调用程序和子程序的程序流程图如图 8.3 所示。程序如下：

```
NAME     EXAMPLE8_7
DSEG     SEGMENT
ARRAYB   DB   （若干个字节数）
COUNT    EQU  $－ARRAYB
ANS      DW   ?
DSEG     ENDS
;
SSEG     SEGMENT  STACK
         DB   80H  DUP（0）
SSEG     ENDS
;
CSEG     SEGMENT
         ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START:   MOV   AX, DSEG
         MOV   DS, AX
         ;
         LEA   SI, ARRAYB
```

```
                MOV    CX, COUNT; 通过约定的寄存器 SI、CX 为子程序提供入口参数;
                CALL   ZNUM
                ;
                MOV    ANS, AX; 通过约定的寄存器 AX 接受子程序提供的出口参数。
                MOV    AH, 4CH
                INT    21H
;
ZNUM        PROC
                XOR    AX, AX
NEXT:       CMP    BYTE   PTR[SI], 0
                JNZ    NZ
                INC    AX
NZ:         INC    SI
                LOOP   NEXT
                RET
ZNUM    ENDP
;
CSEG    ENDS
        END    START
```

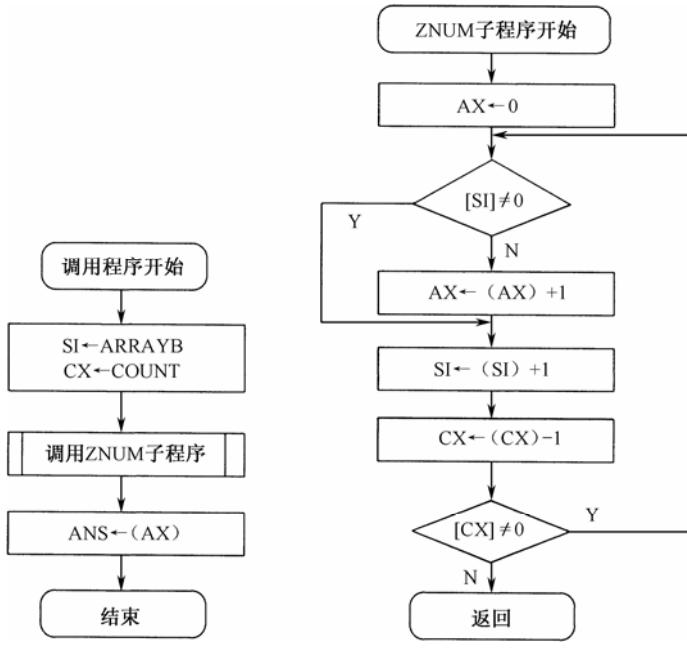


图 8.3　例 8.7 程序流程图

说明：

（1）约定寄存器法的优点是参数传递快，编程较方便，且节省内存单元。

（2）由于寄存器个数很有限，调用程序和子程序中往往要用到一些寄存器，在需传递的参数较多时，会出现寄存器不够用的现象。约定寄存器法只适用于需传递的参数较少的情况。

2. 约定存储单元法

约定存储单元法是指事先约定使用某些存储单元进行入口参数、出口参数的传递。

【例 8.8】编写程序，程序功能与【例 8.7】同，但要求使用约定存储单元法实现参数的传递。

分析：【例 8.7】程序中使用寄存器 SI、CX 传递入口参数，使用寄存器 AX 传递出口参数，以下程序则改用 PARA1、PARA2 单元传递入口参数，改用 PARA3 传递出口参数。调用程序和子程序之间的参数传递可表示为

PARA1←ARRAYB（入口参数）

PARA2←COUNT（入口参数）

PARA3←零元素个数（出口参数）

程序如下：

```
NAME      EXAMPLE8_8
DSEG      SEGMENT
ARRAYB    DB   （若干个字节数）
COUNT     EQU  $－ARRAYB
ANS       DW   ?
PARA1     DW   ?
PARA2     DW   ?
PARA3     DW   ?
DSEG      ENDS
;
SSEG      SEGMENT STACK
          DB   80H  DUP（0）
SSEG      ENDS
;
CSEG      SEGMENT
          ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START:    MOV   AX, DSEG
          MOV   DS, AX
          ;
          LEA   SI,   ARRAYB
          MOV   PARA1, SI
          MOV   PARA2, COUNT; 通过约定的寄存单元 PARA1、PARA2 为子程序提供入口参数;
          CALL   ZNUM
          ;
          MOV   AX, PARA3; 通过约定的寄存单元 PARA3 接受子程序提供的出口参数;
          MOV   ANS, AX
          MOV   AH, 4CH
          INT   21H
          ;
```

```
ZNUM    PROC
        ;
        MOV   SI, PARA1
        MOV   CX, PARA2; 从约定的存储单元 PARA1、PARA2 取入口参数;
        XOR   AX, AX
NEXT:   CMP   BYTE   PTR[SI]   ,0
        JNZ   NZ
        INC   AX
NZ:     INC   SI
        LOOP   NEXT
        MOV   PARA3, AX; 通过约定的存储单元 PARA3 送出口参数。
        RET
ZNUM    ENDP
        ;
CSEG    ENDS
        END   START
```

说明：

（1）约定存储单元法的优点是，每个子程序要处理的数据或送出的处理结果都有独立的存储单元，且传递的参数个数不受限制。

（2）参数传递慢且要占用一定数量的存储单元。约定存储单元法适用于需传递的参数较多的情况。

3. 堆栈法

堆栈法是指通过堆栈进行入口参数、出口参数的传递。

【例 8.9】编写程序，程序功能与【例 8.7】同，但要求使用堆栈法实现参数的传递。

分析：以下程序中，在调用程序的 CALL 指令前将入口参数压入堆栈，在子程序起始处将入口参数弹出堆栈以便子程序使用，即实现入口参数的传递；在子程序的 RET 指令前将出口参数压入堆栈，在调用程序的 CALL 指令后将出口参数弹出堆栈以便调用程序使用。

程序如下：

```
NAME    EXAMPLE8_9
DSEG    SEGMENT
ARRAYB  DB   （若干个字节数）
COUNT   EQU   $－ARRAYB
ANS     DW   ?
DSEG    ENDS
;
SSEG    SEGMENT   STACK
        DB   80H   DUP（0）
SSEG    ENDS
;
```

```
CSEG    SEGMENT
        ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
START:  MOV  AX, DSEG
        MOV  DS, AX
;
        LEA  SI, ARRAYB
        MOV  CX, COUNT
        PUSH  SI
        PUSH  CX; 通过堆栈为子程序提供入口参数;
        CALL  ZNUM
        POP  AX; 通过堆栈接受子程序提供的出口参数;
        MOV  ANS, AX
        MOV  AH, 4CH
        INT  21H
        ;
ZNUM    PROC
        MOV  BP, SP
        MOV  CX, [BP+2]
        MOV  SI, [BP+4] ; 从堆栈取入口参数;
        XOR  AX, AX
NEXT:   CMP  BYTE  PTR[SI], 0
        JNZ  NZ
        INC  AX
NZ:     INC  SI
        LOOP  NEXT
        MOV  [BP+4], AX; 通过堆栈送出口参数。
        RET  2
ZNUM    ENDP
CSEG    ENDS
        END  START
```

在子程序中，从堆栈取入口参数的工作并没有简单地使用指令

```
POP  CX
POP  SI
```

来实现；通过堆栈送出口参数的工作也没有简单地使用指令

```
PUSH  AX
```

来实现。其原因在于，执行 CALL 指令进入子程序后，栈顶内容为返回地址，简单地使用两条 POP 指令不仅不能使子程序得到入口参数，还会给子程序的返回带来问题；另外，若使用 PUSH 指令将出口参数入栈，则接着的 RET 指令会将刚入栈的内容作为返回地址弹出并送 IP，这样显然会出错。

子程序中的指令

```
RET   2
```

实现返回，同时使堆栈指针在返回后再加 2。这样就使得新的栈顶内容为子程序送出的出口参数，以便调用程序通过指令

```
POP   AX
```

接受出口参数。

图 8.4 给出了程序执行中堆栈的变化情况。

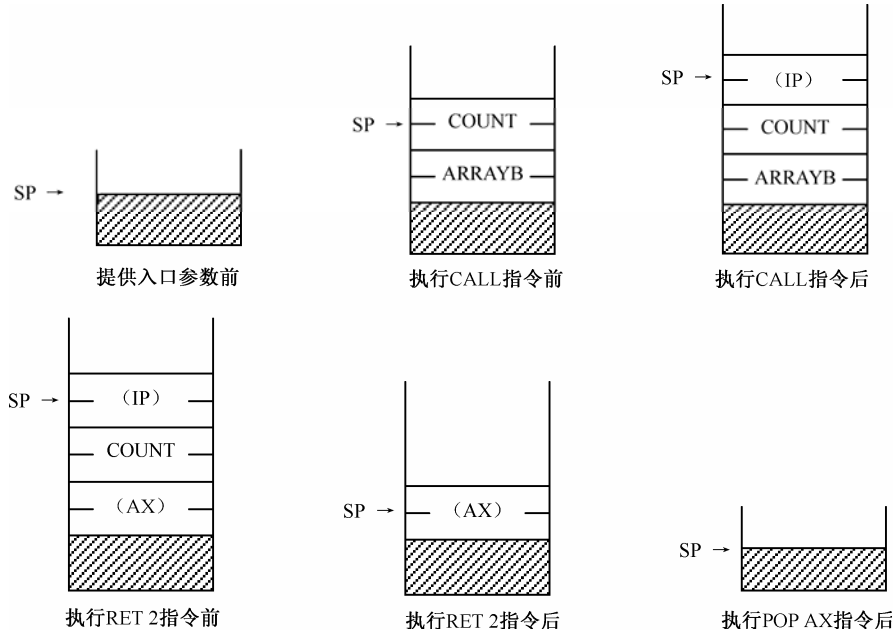


图 8.4　程序执行过程中堆栈的变化情况

说明：

（1）堆栈法的优点是，参数不占用寄存器，也无须开辟专门的内存单元，而是使用公用的堆栈区。在实现入口参数及出口参数的传递后，堆栈恢复原状。

（2）由于参数和子程序的返回地址混在一起，在计算参数在堆栈中的位置时 易于失误。而一旦失误，则在执行返回指令时就可能因栈顶内容不是返回地址而造成返回失败。

按照 8.3.3 中所述，子程序中用到的一些寄存器或内存单元会因为子程序的执行而发生变化，为了避免这种变化对调用程序正常运行的影响，必须进行现场的保护和恢复。

假设有另一个调用程序要调用上述统计字节数组中零元素个数的子程序，而且假设该调用程序的功能决定了需要进行现场的保护和恢复。那么，对于程序 EXAMPLE8_8 而言，需要保护或恢复的有寄存器 AX、SI、CX 及 FLAGS；对于程序 EXAMPLE8_9 而言，则除了寄存器 AX、SI、CX 及 FLAGS 以外，需要保护或恢复的还有 BP。为此 EXAMPLE8_9 中的子程序可以改动如下：

```
ZNUM    PROC
        PUSH   AX
        PUSH   SI
        PUSH   CX
```

```
        PUSH    BP
        PUSHF
        MOV     CX, [BP+10]
        MOV     SI, [BP+12]
          M
        MOV     [BP+12], AX
        POPF
        POP     BP
        POP     CX
        POP     SI
        POP     AX
        RET     2
ZNUM    ENDP
```

程序 EXAMPLE8_7 的子程序中使用的寄存器 AX 该不该保护和恢复？请读者思考。

8.4 程序的嵌套和递归

8.4.1 子程序的嵌套

子程序可以调用另一个子程序，这种调用结构称为子程序的嵌套。由此也可以看出，调用程序、子程序的概念是相对的。图 8.5 表示了两层嵌套的程序结构，而图 8.6 表示了执行各 CALL 指令和 RET 指令后堆栈的情况。

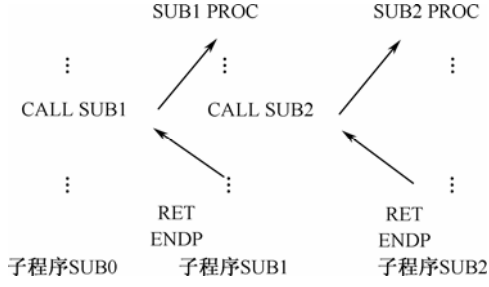


图 8.5 子程序嵌套

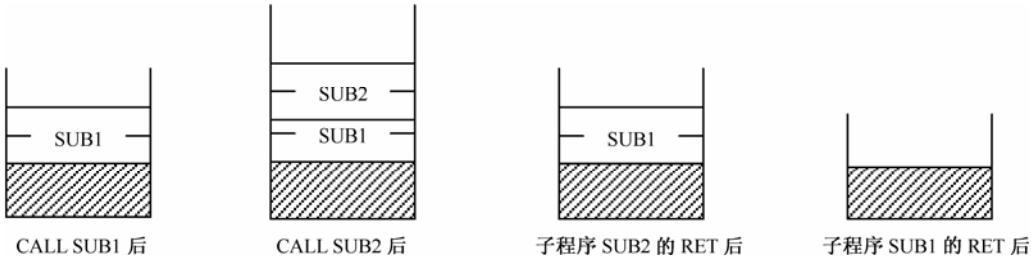


图 8.6 执行各 CALL 指令和 RET 指令后堆栈情况

CALL 指令、RET 指令的功能以及堆栈“先进后出”的特点使得子程序嵌套实现“先调用后返回”，从而使得每一个子程序均能够正常被调用以及正常返回到调用程序。

嵌套子程序在设计中要注意的是，在各层子程序中正确保护和恢复现场，避免各层子程序之间发生寄存器使用的冲突；若使用堆栈法在各层子程序之间进行参数传递，则堆栈中多个返回地址及多个参数混在一起，此时要注意正确计算所用参数的位置，以避免各层子程序返回时出错。

【例 8.10】设 BUF 数据区有两个双字节无符号数，试求两者之和并将其存入 SUM 单元。然后将 SUM 单元的内容以十六进制形式在屏幕上显示出来。

分析：不妨使用图 8.5 所示的子程序嵌套的程序结构。子程序 SUB0 为子程序 SUB1 提供数据区首址并调用 SUB1；子程序 SUB1 实现两个双字节数的相加运算并存储结果，为子程序 SUB2 提供要显示的数据，并且调用 SUB2；子程序 SUB2 实现数据的显示。程序流程图如图 8.7 所示，程序如下：

```
NAME    EXAMPLE8_10
DSEG    SEGMENT
BUF     DW （两个双字节数）
SUM     DW ?
DSEG    ENDS
;
SSEG    SEGMENT   STACK
        DB  80H  DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  DS: DSEG, SS: SSEG, CS: CSEG
SUB0    PROC  FAR
        PUSH  DS
        MOV   AX, 0
        PUSH  AX; 为返回 DOS 作准备;
        MOV   AX, DSEG
        MOV   DS, AX
        LEA   SI, BUF; 通过约定的寄存器 SI 为子程序 SUB1 提供入口参数;
        CALL  SUB1
        RET   ; 返回 DOS;
SUB0    ENDP
;
SUB1    PROC
        MOV   AX, [SI]
        ADD   AX, [SI+2]
        MOV   SUM, AX; 存储结果, 同时通过约定的存储单元 SUM 为子程序 SUB2 提供入
                      口参数;
        CALL  SUB2
```

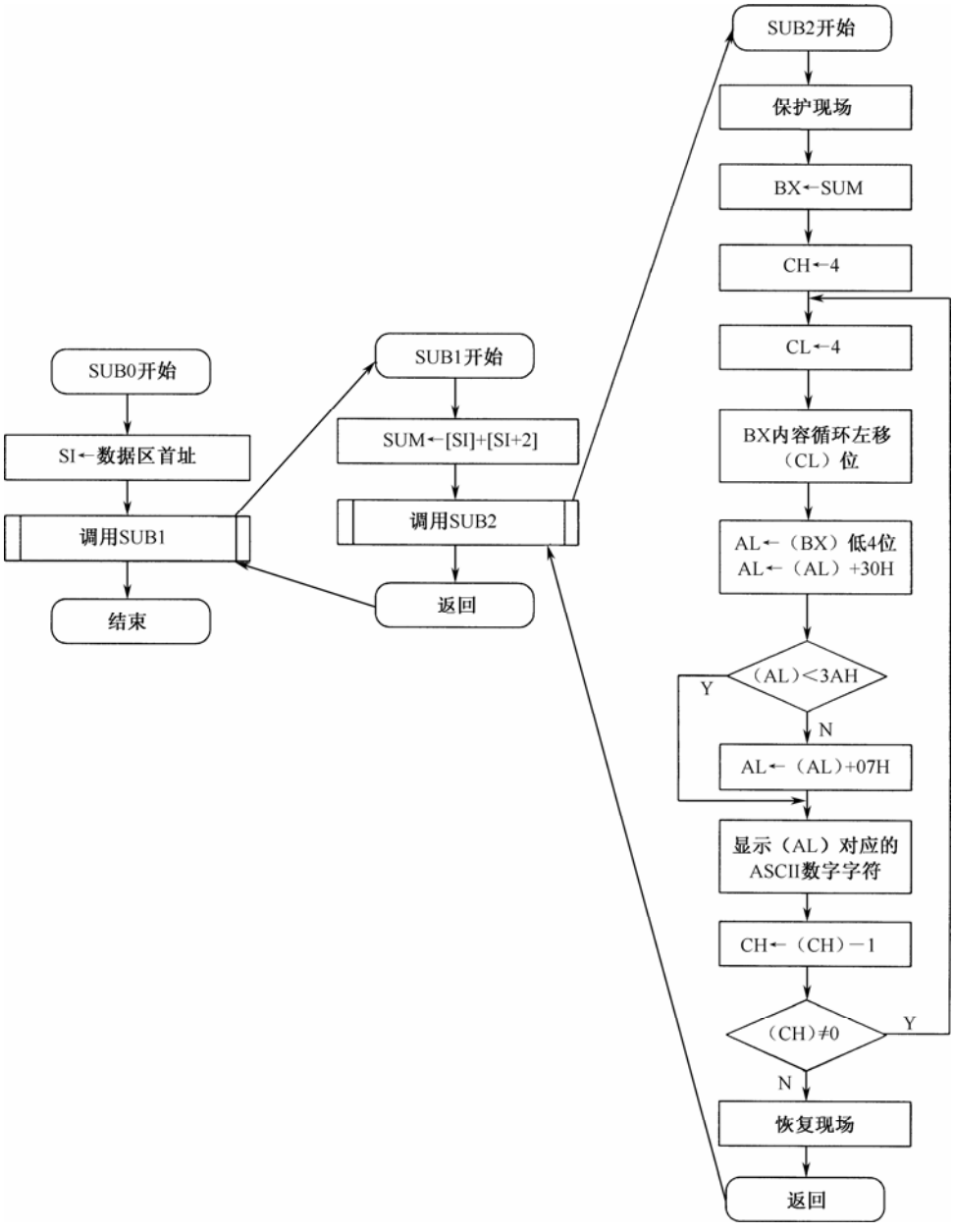


图 8.7　例 8.10 程序流程图以及子程序嵌套结构

```
        RET ; 返回 SUB0;
SUB1    ENDP
;
SUB2    PROC
        PUSH  BX
        PUSH  CX
        PUSH  AX
        PUSH  DX
```

```
        ;
        MOV   BX, SUM
        MOV   CH, 4
        MOV   CL, 4
NEXT:   ROL   BX, CL
        MOV   AL, BL
        AND   AL, 0FH
        ADD   AL, 30H
        CMP   AL, 3AH
        JB    BA

        ADD   AL, 07H
        ;
BA:     MOV   DL, AL
        MOV   AH, 02H
        INT   21H; 显示一位十六进制数;
        DEC   CH
        JNZ   NEXT
        ;
        POP   DX
        POP   AX
        POP   CX
        POP   BX
        ;
        RET; 返回 SUB1。
SUB2    ENDP
        ;
CSEG    ENDS
        END SUB0
```

8.4.2 子程序的递归

子程序调用本身，或通过调用另一个子程序调用本身的结构称为子程序的递归。前者称为直接递归，后者称为间接递归。这样的子程序称为递归子程序。以下子程序采用了直接递归。

```
SUB1    PROC
            M
        CALL SUB1
            M
        RET
SUB1    ENDP
```

以下子程序采用了间接递归。

```
SUB1    PROC
            M
          CALL SUB2
            M
SUB1    ENDP
;
            M
;
SUB2    PROC
            M
```

```
        CALL SUB1
             M
        RET
SUB2    ENDP
;
             M
```

递归这种特殊的嵌套会不会无限地嵌套下去？事实上，在递归子程序设计时必须设置递归结束条件，在每次递归调用中均要对该条件进行判断，递归结束条件不成立则继续递归调用，条件成立则结束递归调用。

【例 8.11】编写计算 *N*!的程序。

在第 7 章中曾使用循环程序解决该计算问题，现要求用递归子程序完成这一计算。假设 $N \leqslant 8$，即 *N*!不超出两字节无符号数的范围。

分析：由阶乘的定义可知：

$$N! = \begin{cases} 1 & N=0 \\ N(N-1)! & N>0 \end{cases}$$

由上式可知，当 $N>0$ 时，要计算 *N*!就必须计算（*N*−1）!；要计算（*N*−1）!就要计算（*N*−2）!；……要计算 1!就要计算 0!。

因此，可以在递归子程序中通过自身调用将 *N*，*N*−1，…，1 依次压入堆栈。这里是否已将 1 压入堆栈就是递归结束条件。一旦递归结束条件满足，则依次将堆栈中的 1，2，…，*N* 出栈相乘，从而算出 *N*!。递归子程序如下：

```
FACT    PROC
        PUSH   DX
        MOV    DX, AX
        CMP    AX, 0; 判断递归结束条件;
        JZ     FR
        DEC    AX
        CALL   FACT; 计算（N−1）!;
        MUL    DX; 计算 N!。
        POP    DX
        RET
FR:     MOV    AX, 1
        POP    DX
        RET
FACT    ENDP
```

8.5 子程序调用与系统功能调用

8.5.1 子程序调用与系统功能调用间的关系

从本章已讲述的内容可以看出，子程序相当于高级语言中的过程和函数。用户可以预先

将经常用到的程序段设计成子程序，在程序的适当位置进行子程序调用。

然而，汇编语言是一种面向机器的语言，与高级语言相比，要求编程者对于硬件有较多的了解。在使用汇编语言进行 I/O 设备、文件系统及内存的管理时，编程工作更显得困难。好在 IBM PC 系统的设计者已经设计出一些具有 I/O 设备、文件系统及内存管理功能的子程序。用户在程序中通过特殊的指令调用这些子程序，就可以较为方便地实现上述管理，这称为系统功能调用。可以看出，系统功能调用是一种特殊的子程序调用。

8.5.2 系统功能调用的方法

系统功能调用分为 DOS 功能调用和 BIOS 功能调用。与用户子程序调用不同，系统功能调用是使用 INT n 指令，n 称为类型号。

DOS 功能调用固定使用 INT 21H 指令。当然，在该指令前要置入口参数，并将功能号送寄存器 AH。本书在 5.2.2 中已经介绍。

BIOS 功能调用与 DOS 功能调用不同，其功能调用指令的类型号有多种。具体调用方法请参阅有关书籍。

注意：

（1）近调用与远调用对堆栈的影响不同。执行近调用指令时，当前的（IP）压入堆栈，而执行远调用指令时，先后入栈的内容分别为（CS）及（IP）。这一点在采用堆栈法进行参数传递时尤其值得注意。例如，对于程序 EXAMPLE8_9 而言，若子程序 ZNUM 与调用程序不在同一个代码段，则原程序中的指令：

```
MOV   CX, [BP+2]
MOV   SI, [BP+4]
MOV   [BP+4], AX
```

就应分别改为：

```
MOV   CX, [BP+4]
MOV   SI, [BP+6]
MOV   [BP+6], AX
```

（2）要搞清保护、恢复现场与参数传递之间的关系。对于在子程序执行过程中内容可能发生变化的寄存器或存储单元，该不该在子程序起始处加以保护，在子程序的 RET 指令前予以恢复？这要视具体问题而定。对于一个具体问题而言，若调用程序要求在调用子程序后这些寄存器或存储单元内容不变，就应该将它们作为现场进行保护和恢复；若调用程序正是需要使用调用子程序后这些寄存器或存储单元变化后的内容，则它们的作用是进行出口参数的传递，在这种情况下将它们作为现场进行保护和恢复就达不到预期目的。如程序 EXAMPLE8_7 的子程序中的寄存器 AX 就不能作为现场进行保护和恢复。

（3）为了便于其他程序的调用，在编制子程序时应以注释方式给出说明。说明的内容包括以下方面。

- 子程序名；
- 子程序功能；
- 入口参数及传送方式；
- 出口参数及传送方式；
- 子程序中内容可能发生变化的寄存器及存储单元；

- 子程序本身需调用的子程序名；
- 典型示例。例如，对于程序 EXAMPLE8_2，可以在其中的子程序前加上如下说明：
- 子程序名：SUB1；
- 功能：计算 X!（结果不超过 4 字节无符号数范围）；
- 入口参数：ECX←X；
- 出口参数：由 EAX 提供；
- 子程序中内容可能发生变化的寄存器：ECX，EAX，EFLAGS；
- 子程序本身需调用的子程序：无；
- 典型示例：

入口参数：ECX←9；
出口参数：（EAX）=362880。

8.6 子程序设计综合举例

【例 8.12】输入一串字符，分别统计其中字母、数字及其他字符的个数，并在显示器上输出。

分析：根据题目要求，需要实现的功能包括：

（1）输入一串字符。可以使用特殊的子程序调用——DOS 系统功能调用实现，具体是采用 1 号 DOS 系统功能调用。由于字符个数未知，可以实施循环调用，并以 Enter 符作为结束标志。

（2）统计其中字母、数字及其他字符的个数。可以在字符输入后即做出判断，并用 3 个寄存器分别存放这 3 类字符的个数。这里假设输入时“Cape Lock”处于打开状态，即输入的字母为大写形式。

（3）显示这串字符中字母、数字及其他字符的个数。可通过 3 次调用显示子程序实现。在显示子程序中，某类字符个数的显示又可以使用特殊的子程序调用——DOS 系统功能调用实现，具体是采用 2 号 DOS 系统功能调用。

程序流程图如图 8.8 所示，程序如下。

```
NAME    EXAMPLE8_12
CSEG    SEGMENT
        ASSUME   CS: CSEG
START:  XOR   BL, BL              ; BL 用于统计数字字符个数
        XOR   CH, CH              ; CH 用于统计字母字符个数
        XOR   CL, CL              ; CL 用于统计其他字符个数
AGAIN:  MOV   AH, 1
        INT   21H
        CMP   AL, 0DH
        JZ    EXIT
        CMP   AL, '0'
        JB    OTHER
        CMP   AL, '9'
```

```
            JA    NEXT
            INC   BL
            JMP   AGAIN
NEXT:       CMP   AL, 'A'
            JB    OTHER
            CMP   AL, 'Z'
            JA    OTHER
            INC   CH
            JMP   AGAIN
OTHER:      INC   CL
            JMP   AGAIN
EXIT:       MOV   DL, BL
            CALL  DISP
            MOV   DL, CH
            CALL  DISP
            MOV   DL, CL
            CALL  DISP
DISP        PROC
            ADD   DL, 30H
            MOV   AH, 2
            INT   21H
            RET
DISP        ENDP
CSEG        ENDS
            END   START
```

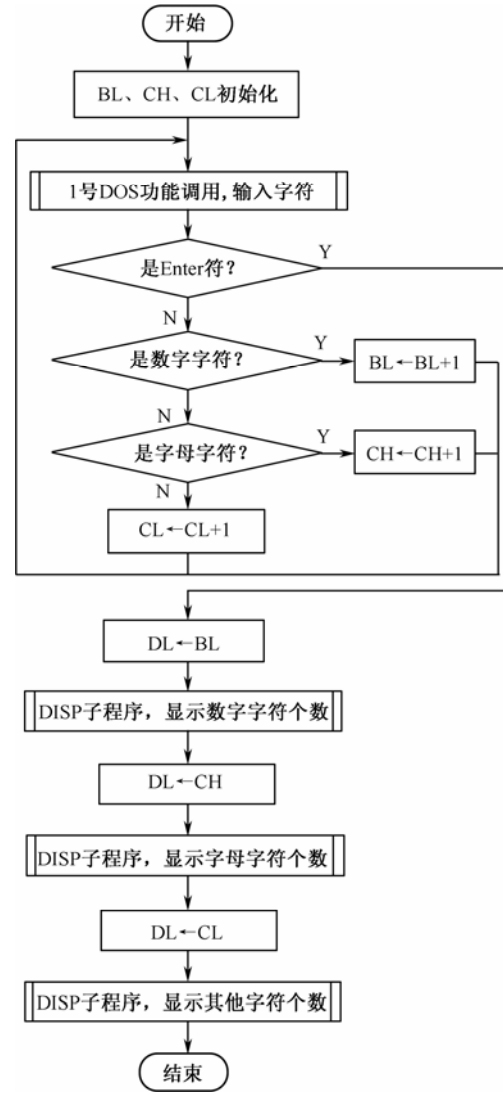


图 8.8　例 8.12 程序流程图

【例 8.13】输入某班一门课程的考试成绩，并存放到 ASM 存储区，将最高分存放到变量 MAX，并在显示器上输出。设该班人数为 30，考试成绩不超过 99 分。

分析：根据题目要求，需要实现的功能包括：

（1）输入 30 个两位数。每个两位数的输入可以分别通过一键盘输入子程序实现，在键盘输入子程序中，可通过 7 号 DOS 系统功能调用实现。

（2）将输入的两位数由 ASCII 码转换为压缩的 BCD 码，可通过 ASCII 到 BCD 转换子程序实现。

（3）产生最高分。可通过求最高得分子程序实现。

（4）显示最高分。首先需要将压缩的 BCD 码形式的最高分转换成两个 ASCII 码，这可以通过 BCD 到 ASCII 转换子程序实现。然后通过一显示子程序实现，在显示子程序中，可以通过 2 号 DOS 系统功能调用实现。

程序流程图如图 8.9 所示，程序如下：

```
NAME    EXAMPLE8_13
DSEG    SEGMENT
ASM     DB   30 DUP （？）
MAX     DB   ?
DSEG    ENDS
;
SSEG    SEGMENT   STACK
        DB   80H   DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
        ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
; 主程序
START:  MOV   AX, DSEG
        MOV   DS, AX
        LEA   DI, ASM
        MOV   CH, 30
AGAIN:  CALL   KEYIN          ; 调用键盘输入子程序
                                以输入十位数
        MOV   BL, AL
        CALL   KEYIN          ; 调用键盘输入子程序
                                以输入个位数
        CALL   A_B            ; 调用 ASCII→BCD 转
                                换子程序以生成压缩
                                的 BCD 码
        MOV   [DI], AL        ; 压缩的 BCD 码存
                                入 ASM 存储区
        INC   DI
        DEC   CH
        JNZ   AGAIN
;
        LEA   DI, ASM
        MOV   CX, 30
        CALL   CMAX           ; 调用求最高得分子程序
        CALL   B_A            ; 调用 BCD→ASCII 子程序
        CALL   DISP           ; 调用显示子程序
        MOV   AH, 4CH
        INT   21H

; 键盘输入子程序
```

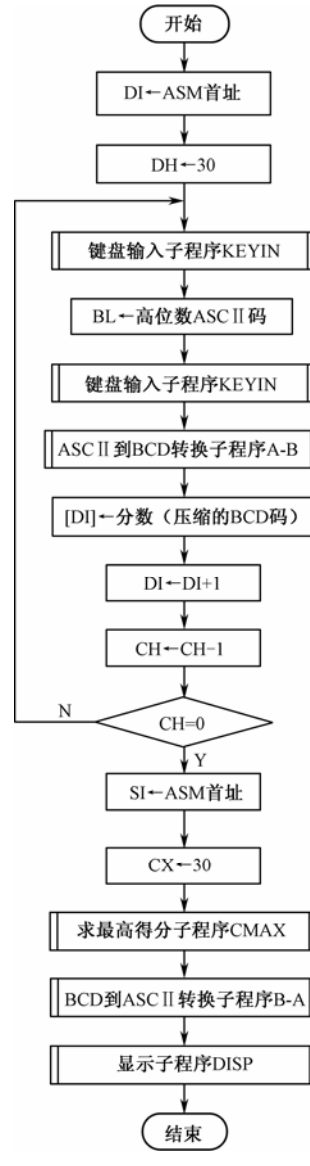


图 8.9　例 8.12 程序流程图

```
KEYIN    PROC
         MOV   AH, 7
         INT   21H
         RET
KEYIN    ENDP

;  ASCII→BCD 转换子程序
A_B      PROC
         MOV   CL, 4
         SHL   BL, CL
         SUB   AL, 30H
         ADD   AL, BL
         RET
A_B      ENDP

;  求最高得分子程序
CMAX     PROC
         DEC   CX
         MOV   AL, [SI]
L1:      CMP   AL, [SI+1]
         JA    NEXT
         MOV   AL, [SI+1]
NEXT:    INC   SI
         LOOP  L1
         MOV   MAX, AL
         RET
CMAX     ENDP

;  BCD→ASCII 转换子程序
B_A      PROC
         MOV   AL, MAX
         MOV   BL, AL
         AND   AL, 0F0H
         MOV   CL, 4
         SHR   AL, CL
         ADD   AL, 30H
         AND   BL, 0FH
         ADD   BL, 30H
         RET
B_A      ENDP

;  显示子程序
DISP     PROC
         MOV   DL, AL
         MOV   AH, 2
         INT   21H
         MOV   DL, BL
         MOV   AH, 2
         INT   21H
         RET
DISP     ENDP

CSEG     ENDS
         END   START
```

习　　题

8.1　指出 CALL 指令与 JMP 指令的异同。

8.2　试述远过程中 RET 指令的功能。

8.3　分析程序并指出程序执行结果

（1）

```
DSEG     SEGMENT
STRS     DB   'S12TR4IN63G$'
COUNT    EQU  $-STRS
STRD     DB   COUNT DUP（？）
DSEG     ENDS
;
CSEG     SEGMENT
         ASSUME  DS: DSEG, ES: DSEG, CS: CSEG
```

```
START:  MOV   AX, DSEG
        MOV   DS, AX
        MOV   ES, AX
        LEA   SI, STRS
        LEA   DI, STRD
        CLD
NEXT:   LODSB
        CMP   AL, '$'
        JZ    EXIT
        CALL  JUG
        JNC   NEXT
        STOSB
        JMP   NEXT
EXIT:   MOV   AH, 4CH
        INT   21H
JUG     PROC
        CMP   AL, '0'
        JB    CHAR
        CMP   AL, '9'
        JA    CHAR
        CLC
        RET
CHAR:   STC
        RET
JUG     ENDP
CSEG    ENDS
        END   START
```

程序执行后，STRD 存储区内容为_____。

（2）

```
DSEG    STGMENT
X       DB   3
Y       DW   ?
DSEG    ENDS
;
SSEG    SEGMENT
        DB   80H  DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
        ASSUME  DS: DSEG, SS:
        SSEG, CS: CSEG
START:  MOV   AX, DSEG
        MOV   DS, AX
        MOV   AL, X
        CALL  FX
        MOV   DX, AX
        MOV   AL, X
        IMUL  AL
        CALL  FX
        SUB   AX, DX
        MOV   Y, AX
        MOV   AH, 4CH
        INT   21H
;
FX      PROC
        PUSH  DX
        MOV   DL, AL
        MOV   AL, 3
        IMUL  DL
        ADD   AX, 4
        XCHG AX, DX
        CBW
        IMUL  DX
        DEC   AX
        POP   DX
        RET
FX      ENDP
CSEG    ENDS
        END   START
```

此程序执行后 Y＝_____。

8.4　以下程序的功能是将一个字节的二进制数转换成二进制数的 ASCII 码（0 转换为 '0'，1 转换为 '1'）。试根据给定的程序功能填写方框中的指令。

```
DSEG    SEGMENT
BIN     DB   11100011B
ASCB    DB   8   DUP（？）
DSEG    ENDS
;
SSEG    SEGMENT   STACK
        DB   80H   DUP（0）
SSEG    ENDS
;
CSEG    SEGMENT
        ASSUME   DS: DSEG, SS: SSEG, CS: CSEG
START:  MOV   AX, DSEG
        MOV   DS, AX
        MOV   AH, BIN
        PUSH   AX
        LEA   DI, ASCB
        PUSH   DI
        MOV   AX, 8
        PUSH   AX
        CALL   BTOA
        MOV   AH, 4CH
        INT   21H
        ;
BTOA    PROC
        PUSH   DI
        PUSH   CX
        PUSH   DX
        MOV   BP, SP
        [          ]
        [          ]
        [          ]
NEXT:   ROL   DX, 1
        MOV   AL, DL
        AND   AL, 1
        ADD   AL, 30H
        MOV   [DI], AL
        INC   DI
        LOOP   NEXT
        POP   DX
        POP   CX
```

```
        POP   DI
        
BTOA    ENDP
CSEG    ENDS
        END     START
```

8.5　根据程序说明的功能，指出下列程序中的错误。

```
; 子程序名: FUNC
; 功能: 计算 X*Y（结果不超过 2 字节无符号数范围）
; 入口参数: AH←X, AL←Y
; 出口参数: 由 DX 提供
; 子程序中内容可能发生变化的寄存器: AX, DX, FLAGS
; 子程序本身需调用的子程序: 无
; 典型示例:
  入口参数: AH←20H, AL←11H
  出口参数: （DX）=0220H
FUNC    PROC
        PUSH  AX
        PUSH  DX
        PUSHF
        MUL   AH
        MOV   DX, AX
        POP   AX
        POP   DX
        POPF
        ENDP
```

8.6　分别使用约定寄存器法、约定存储单元法改写 8.4 题中的程序。

8.7　编写程序，将以 BLKS 为首址的连续 *N* 个字节数传送至以 BLKD 为首址的存储区。要求用子程序实现数据的传送，由调用程序根据 BLKS、BLKD 两者的位置关系以及数据块的大小为子程序提供入口参数。

8.8　编写一个递归程序，完成自然数 1～100 的求和运算。

第 9 章　高级汇编语言技术

本章将介绍宏汇编，重复汇编，条件汇编及库的使用等高级汇编语言技术。在掌握前面各章所述的基本汇编语言技术的基础上掌握本章内容，将有利于简化源程序，减少目标代码，缩短程序执行时间，即有利于编写简洁高效的汇编语言程序。

9.1　宏汇编

上一章所介绍的子程序结构具有不少优点，比如可避免编制程序中的重复劳动，节省程序代码所占的存储空间，有利于模块化程序设计等。但是，使用这种结构也存在一些缺点，比如执行子程序调用与子程序返回指令，保护和恢复现场，提供入口参数和出口参数等环节将花费程序执行时间和存储空间。在子程序较小而需保护和恢复的寄存器、存储单元较多，涉及的参数较多时，这种开销就更为突出。IBM-PC 宏汇编所提供的宏汇编技术可以较好地解决这一问题，宏汇编技术包括宏定义与宏调用。

9.1.1　宏定义

1. 宏定义的格式

宏定义即宏指令的定义，其一般格式为：

```
宏指令名 MACRO [形参表]
            宏体
        ENDM
```

说明：

（1）宏指令名由编程序者自定，但必须符合标号的命名规则。

（2）MACRO 和 ENDM 是一对伪指令，分别表示宏定义的开始和结束。

（3）宏体可以是指令、伪指令及宏指令构成的程序段。

（4）形参表可根据需要作取舍。当需要设置多个形参时，各形参之间要用逗号分隔。

2. 宏定义的功能

宏定义的功能在于，将宏体定义为一条宏指令，以便在其后的程序中通过对宏指令的调用来使用对应的宏体。

【例 9.1】以下宏定义所定义的宏指令 AX10 可以实现寄存器 AX 内容乘以 10 的功能。

```
AX10   MACRO
PUSH   DX
SAL    AX, 1
MOV    DX, AX
SAL    AX, 1
SAL    AX, 1
```

```
ADD   AX, DX
POP   DX
ENDM
```

【例 9.2】以下宏定义所定义的宏指令 MUL10 可以实现 16 位通用寄存器（除 DX）或 16 位存储单元内容乘以 10 的功能。该宏定义设置了一个形参 X，乘以 10 的操作从形式上来说是针对 X 进行的。

```
MUL10   MACRO X
PUSH   DX
SAL   X, 1
MOV   DX, X
SAL   X, 1
SAL   X, 1
ADD   X, DX
POP   DX
ENDM
```

9.1.2 宏调用和宏扩展

1. 宏调用的格式

宏调用的格式为

宏指令名 [实参表]

说明：

（1）宏指令名所指定的宏指令的定义必须放在该宏调用之前。

（2）实参表通常与宏定义中形参表相对应。当需要使用多个实参时，各实参之间要用逗号分隔。

2. 宏扩展

汇编程序在对源程序作汇编时，若遇到宏调用，即遇到源程序中的宏指令，则将对应的宏体取代该宏指令，宏体中的形参则使用宏调用提供的对应实参来取代。这一过程就称为宏扩展。用列表文件查看源程序时，将看到宏扩展所产生的各条指令前加有“+”标记。

【例 9.3】在已经设置了【例 9.1】及【例 9.2】中的宏定义的后，以下宏调用及其宏扩展情况如下：

设有宏调用：

```
AX10
        M
MUL10 BX
        M
MUL10 BUF
        M
```

则对应的宏扩展为

```
          M
+ PUSH   DX
+ SAL    AX, 1
+ MOV    DX, AX
+ SAL    AX, 1
+ SAL    AX, 1
+ ADD    AX, DX
+ POP    DX; 将寄存器 AX 内容乘以 10;
          M
+ PUSH   DX
+ SAL    BX, 1
+ MOV    DX, BX
+ SAL    BX, 1
+ SAL    BX, 1
+ ADD    BX, DX
+ POP    DX; 将寄存器 BX 内容乘以 10;
          M
+ PUSH   DX
+ SAL    BUF, 1
+ MOV    DX, BUF
+ SAL    BUF, 1
+ SAL    BUF, 1
+ ADD    BUF, DX
+ POP    DX;      将字变量 BUF 内容乘以 10;
          M
```

9.1.3 宏定义和宏调用中参数的使用

宏定义和宏调用可以不使用参数，如【例 9.1】中的宏定义及【例 9.3】中的第一个宏调用所示。但是，在宏定义和宏调用中使用参数时，将显得更加灵活。如【例 9.2】中的宏定义使用了形参 X，【例 9.3】中第二个宏调用和第三个宏调用分别使用实参 BX 和 BUF 取代形参 X，从而分别实现了将寄存器 BX 和字变量 BUF 内容乘以 10 的功能。可见，使用参数的宏指令 MUL10 要比不使用参数的宏指令 AX10 更灵活。

在使用参数时，有以下规定：

（1）实参与形参的个数可以不一致。一般情况下，宏调用中实参的个数与宏定义中形参的个数一致，在宏扩展时，宏体中的形参用宏调用提供的对应实参来取代。但是，汇编程序并不限定两者个数必须一致。

若实参个数大于形参个数，则多余的实参被忽略；若形参个数大于实参个数，则多余的形参做“空”处理。

（2）形参可以作为宏体中指令的助记符、操作数及标号，宏调用用实参取代形参时必须保证所产生的指令是有效的。

【例 9.4】以下宏定义中的第一个形参用做指令助记符，其余形参用做操作数。

```
MM    MACRO    OP, X, Y
      PUSH    AX
      MOV    AX, X
      OP    AX, Y
      MOV    X, AX
      POP    AX
      ENDM
```

设有宏调用

```
      MM    ADD, [BX], [SI]
```

则其宏扩展为

```
+ PUSH    AX
+ MOV    AX, [BX]
+ ADD    AX, [SI]
+ MOV    [BX], AX
+ POP    AX
```

设有宏调用

```
MM    SUB , DATA1, DATA2
```

则其宏扩展为

```
+ PUSH    AX
+ MOV    AX, DATA1
+ SUB    AX, DATA2
+ MOV    DATA1, AX
+ POP    AX
```

若变量 DATA1 或 DATA2 没有事先定义为字变量，则上述宏扩展就会产生无效的指令，该宏调用就会出错。

（3）宏运算符&、＜＞、%及!的使用。

① 形参可以作为宏体中指令助记符、操作数或标号的某一部分，在宏体中必须使用“&”将形参与其余部分连接起来，以免宏调用时使得对应的实参与其余部分相分离。

【例 9.5】以下宏定义中的第一个形参 CON 用做指令助记符的一部分，故在宏体中用“&”将“JN”与该形参相连接。

```
CONCALL    MACRO    CON, SUBN
                    LOCAL    GOON; LOCAL 伪指令的功能见后
                    JN&CON GOON
                    CALL SUBN
GOON:       NOP
                   ENDM
```

该宏定义所定义的宏指令 CONCALL 可以实现子程序的条件调用。

设有宏调用

```
CONCALL C, SUB1
```

则其宏扩展为：

```
+ JNC ??0000
+ CALL SUB1
+ ??0000: NOP
```

其功能为当 CF 为 1 则调用子程序 SUB1，否则不作调用。在宏调用时，实参“C”取代形参“CON”且紧接“JN”之后形成指令助记符。宏扩展中的??0000 为汇编程序自动产生的标号。

② 当某一个实参中含有空格，逗号等分隔符时，必须用“<>”将其括起，以免被视作多个实参。

【例 9.6】源程序一般都要定义堆栈段，各个源程序所使用的定义格式基本相同，只是对堆栈段的大小等有不同要求。为此可以作以下宏定义，将其放入库中，以便各源程序调用。

```
STK    MACRO X
SSEG SEGMENT STACK
        DB X
SSEG ENDS
STK    ENDM
```

若某个源程序需要定义 200 个字节，初值为 0 的堆栈段，则可以使用宏调用：

```
STK <200 DUP (0)>
```

其宏扩展为：

```
+SSEG SEGMENT STACK
+       DB 200 DUP（0）
+SSEG ENDS
```

由于宏调用时将可能有空格的实参用“<>”括起，汇编程序将“200 DUP（0）”作为一个实参来取代形参 X。

若不使用“＜＞”，使用宏调用：

```
STK 200 DUP（0）
```

则宏扩展为

```
+ SSEG    SEGMENT STACK
+             DB 200
+ SSEG ENDS
```

③ 在宏调用时，若要求将实参所代表的数值（而不是实参本身）替代形参，则必须在实参前使用“%”。

【例 9.7】设有以下宏定义：

```
DISP MACRO X
      DB 'ANSWER: ', '&X', '$'
DISP    ENDM
```

则宏调用

```
      DISP   %（2*11-8）
```

产生的宏扩展为

```
      + DB 'ANSWER: ', '14', '$'
```

不使用符号“%”的宏调用

```
DISP   2*11-8
```

产生的宏扩展却是

```
+DB 'ANSWER: ', '2*11-8',  '$'
```

④ 当需要在实参中使用“&”、“＜”、“＞”、“%”等符号，但不做宏运算符时，就必须在其前使用“!”。

【例 9.8】在已设置【例 9.7】中的宏定义的情况下，宏调用

```
DISP ! %（2*11-8）
```

产生的宏扩展为

```
+DB 'ANSWER: ', '%（2*11-8）', '$'
```

宏调用中的“%”前使用了宏运算符“!”，故“%”被看做一个符号，而失去了宏运算符的功能。

（4）宏定义中标号和变量的处理。宏指令一经定义便可在源程序中调用，若宏体中使用了标号或变量，在多次宏调用时就会出现多个相同标号或出现变量的重复定义，使用 LOCAL 伪指令可以解决这一问题。

LOCAL 伪指令的使用方法及功能如下：

① LOCAL 伪指令的一般格式：

LOCAL 标号及变量表

各标号、变量之间用逗号分隔。

② 在宏定义中，LOCAL 伪指令必须紧接 MACRO 伪指令之后。

③ 在处理各个宏调用时，汇编程序将自动以？？0000，？？0001，…？？FFFF 替代 LOCAL 伪指令列出的各个标号或变量，从而避免多次宏调用时出现多个相同标号或出现变量重复定义的问题。

【例 9.9】在已设置【例 9.5】中的宏定义后，宏调用：

```
CONCALL C, SUB1
          M
CONCALL Z, SUB2
```

产生以下宏扩展：

```
+JNC ??0000
+CALL SUB1
+??0000: NOP
    M
+JNZ ??0001
+CALL SUB2
+??0001: NOP
    M
```

对于宏调用

```
CONCALL C, SUB1
```

汇编程序??0000 取代宏体中的标号 GOON；对于宏调用

```
CONCALL Z, SUB2
```

汇编程序用??0001 取代宏体中的标号 GOON，从而避免了宏扩展后的程序中两处定义同一标号 GOON。

9.1.4 宏嵌套

宏嵌套包括两种情况：其一，宏定义的宏体中包括宏调用，即在宏体中调用宏指令。在这种情况下要注意，其中调用的宏指令必须先行定义；其二，宏体中包括宏定义。在这种情况下要注意，不能在源程序中直接调用内层定义的宏指令。换言之，在源程序中只有通过外层宏指令的调用才能调用内层宏指令。

说明：

（1）宏指令名可以与指令助记符及伪指令名同名。在此情况下，宏指令的优先级较高，同名的指令或伪指令的原有功能失效。在利用这一方法改变了某个指令助记符或伪指令名的原有功能后，可以通过宏调用来使用新定义的功能。若要恢复其原有功能，只要使用清除宏定义的伪指令：

PURGE　宏指令名表

【例 9.10】以下宏定义将指令 CBW 的功能改为，将寄存器 BL 中的有符号数的符号扩展到寄存器 BH 中。

```
CBW MACRO
    LOCAL P
      XOR BH, BH
      TEST BL, 80H
      JZ P
      MOV BH, 0FFH
P:    NOP
      ENDM
```

在设置了上述宏定义后，以下程序段中的两条 CBW 指令具有不同的功能：

```
      N
CBW; 宏调用, 将（BL）的符号扩展到 BH;
      N
PURGE CBW; 清除对 CBW 的宏定义;
CBW; 将（AL）的符号扩展到 AH。
```

（2）宏定义时也要注意现场的保护和恢复。例如，【例 9.1】中的宏定义 PUSH DX 和 POP DX 指令实现现场的保护和恢复，以免调用宏指令 AX10 后寄存器 DX 的内容被破坏。

（3）宏汇编与子程序的比较如表 9.1 所示。

表 9.1　宏汇编与子程序的比较

	宏　汇　编	子　程　序
目标代码所占空间	有几次宏调用就有几次宏扩展，故并不简化目标代码	子程序目标代码只出现一次，故目标代码短
程序运行速度	无须转返，程序运行速度快	需要转返工作，程序运行速度慢
处理时机	在汇编时由汇编程序实现宏扩展。CPU 执行的是经过宏扩展的程序	在执行时，CPU 通过转子指令执行子程序中的指令
适用场合	程序运行速度是主要考虑因素	目标代码所占空间是主要考虑因素

9.2 重复汇编

在解决实际问题时，有时需要连续重复编写一组相同或基本相同的指令或伪指令。针对这组指令或伪指令进行宏定义，然后通过连续重复宏调用可以避免重复编写的问题。然而，对于这种指令或伪指令组的连续重复问题，使用重复汇编结构则更为简便。

重复汇编结构有三种，分别使用伪指令 REPT、IRP 和 IRPC 实现。使用这三条伪指令的重复汇编结构的区别在于重复次数如何确定，以及对重复的指令或伪指令组如何作规律性的变化。

9.2.1 使用REPT伪指令的重复汇编结构

格式：REPT 整数表达式

　　重复体

　ENDM

功能：使汇编程序对重复体作重复汇编，以整数表达式的值作为重复次数。

说明：

（1）重复体由指令、伪指令及宏指令组成。

（2）整数表达式中各项必须预先定义，以便在汇编时能够得到一个具体的整数。

【例 9.11】设有重复汇编结构如下：

```
CHAR=41H
REPT 10
    DB CHAR
    CHAR=CHAR+1
ENDM
```

汇编程序在汇编时将对重复体

```
DB CHAR
CHAR=CHAR+1
```

重复汇编 10 次，即汇编为

```
CHAR=41H
DB CHAR; CHAR 为 41H
CHAR=CHAR+1
DB CHAR; DB 为 42H
CHAR=CHAR+1
    M
DB CHAR; CHAR 为 4AH
CHAR=CHAR+1
```

其结果等价于：

```
DB 41H, 42H, 43H, 44H, 45H, 46H, 47H, 48H, 49H, 4AH
```

【例 9.12】使用该重复汇编结构可以在连续的 81 个字节单元中存放九九乘法表的数值：

```
N=0
  REPT 9
        N=N+1
        M=0
         REPT 9
              M=M+1
              DB M*N
         ENDM
  ENDM
```

这实际上是一个嵌套的重复汇编结构。内外两层重复汇编结构均使各自的重复体重复 9 次。汇编后等价于：

```
DB 1, 2, 3, …9
DB 2, 4, 6, …18
DB 3, 6, 9, …27
          M
DB 9, 18, 27, …81
```

9.2.2 使用IRP伪指令的重复汇编结构

格式：IRP 形参，〈实参表〉

　　重复体

　ENDM

功能：使汇编程序对重复体作重复汇编，每作一次汇编就依次将实参表中的一个实参取代重复体中的形参。显然，重复次数就等于实参表中实参的个数。

说明：

（1）重复体的规定同前，而且重复体中含有形参。

（2）实参表中的各个实参须用逗号分隔，实参取代形参后得到的应是有效的指令、伪指令或宏指令。

【例 9.13】设有重复汇编结构如下：

```
IRP REG,〈AX,BX,CX,DX〉
    PUSH REG
ENDM
```

汇编程序在汇编时将对重复体

```
PUSH  REG
```

重复汇编 4 次，且分别用实参 AX，BX，CX 及 DX 取代形参 REG。其结果等价于：

```
PUSH  AX
PUSH  BX
PUSH  CX
PUSH  DX
```

【例 9.14】使用以下重复汇编结构，可以方便地建立 0～9 的立方值表。

```
IRP N, <0, 1, 2, 3, 4, 5, 6, 7, 8, 9>
  DB N*N*N
ENDM
```

9.2.3 使用IRPC伪指令的重复汇编结构

格式：IRPC 形参，字符串

重复体

ENDM

功能：使汇编程序对重复体作重复汇编，每作一次汇编就依次用字符串中的一个字符取代重复体中的形参。显然，重复次数就等于字符串中字符的个数。

说明：当重复体中的形参只代表指令助记符、操作数或标号等内容的一部分时，要使用“&”将其与其余部分连接起来，以免用字符取代形参时使该字符与其余部分相分离。

【例 9.15】设有重复汇编结构如下：

```
IRPC X, 0123456789
    DB X
ENDM
```

汇编程序在汇编时将对重复体

```
DB X
```

重复汇编 10 次，且分别用 0，1，2，…，9 取代形参 X。其结果等价于

```
DB 0, 1, 2, …, 9
```

【例 9.16】用以下重复汇编结构可以实现【例 9.13】所述的功能。

```
IRPC REG, ABCD
  PUSH REG&X
  ENDM
```

说明：

（1）对于需多次重复相同或基本相同程序段的源程序，使用宏汇编或重复汇编方法均可达到简化程序的目的。但是，宏汇编适用于非连续重复的场合，而重复汇编适用于连续重复的场合。

（2）就重复实现某些操作这一点而言，重复汇编与循环程序结构均可达到简化程序的目的。但两者有着多方面的区别，如表 9.2 所示。

表 9.2 重复汇编与循环程序结构的比较

	重 复 汇 编	循 环 程 序
目标代码所占空间	重复体将重复指定的次数，故并不简化目代码	重复部分的目标代码只出现一次，故目标代码短
程序运行速度	无须循环控制，程序运行速度快	需要循环控制，程序运行速度慢
处理时机	在汇编时由汇编程序对重复体作重复汇编。CPU 执行的是经过重复汇编的各个重复体目标代码	在执行时，CPU 在循环控制指令的控制下确定是否重复执行循环体
灵活性	重复体可以包括指令、伪指令及宏指令；重复汇编所得到的各个重复体目标代码可以完全相同也可以有所区别。此法较灵活	循环体只能是指令或宏指令，但不能是伪指令；每次重复执行的目标代码完全相同。此法相对说来欠灵活
适用场合	程序执行速度是主要考虑因素，需重复的部分有伪指令，以及各个重复体目标代码需要有所区别的场合	目标代码所占空间是主要考虑因素，需重复的部分不含伪指令以及每次执行的目标代码完全相同的场合

9.3 条件汇编

9.3.1 条件汇编的概念及条件汇编的结构

汇编程序可以根据条件来确定是否汇编某段源程序，也就是说可以实现条件汇编。IBM-PC 宏汇编系统提供了一组条件伪指令，使用这些条件伪指令就可以构成条件汇编结构。条件汇编结合宏指令使用可以更好地发挥作用。

条件汇编结构的格式如下：

（1）IFXX　条件
　　程序块 1
　ELSE
　　程序块 2
　ENDIF

（2）IFXX　条件
　　程序块
　ENDIF

条件汇编结构的功能：

格式（1）所示结构的功能为：使汇编程序对条件进行判断，条件成立则汇编程序块 1，并接着汇编 ENDIF 后的源程序段；条件不成立则汇编程序块 2，并接着汇编 ENDIF 后的源程序段。

格式（2）所示结构的功能为：使汇编程序对条件进行判断，条件成立则汇编程序块，并接着汇编 ENDIF 后的源程序段；条件不成立则直接汇编 ENDIF 后的源程序段。

9.3.2 条件汇编伪指令

条件汇编伪指令中 IF 伪指令有多种形式，如表 9.3 所示。

表 9.3　条件汇编伪指令

伪　指　令	汇 编 条 件
IF 表达式	表达式≠0
IFE 表达式	表达式=0
IFDEF 符号	符号已被定义，或已用 EXTRN 伪指令说明为外部符号
IFNDEF 符号	符号未被定义，也未用 EXTRN 伪指令说明为外部符号
IFB ＜参数＞	参数为空
IFNB ＜参数＞	参数非空
IFIDN ＜字符串 1＞,＜字符串 2＞	字符串 1 与字符串 2 相同
IFDIF ＜字符串 1＞，＜字符串 2＞	字符串 1 与字符串 2 不同
IF1	第一次扫描
IF2	第二次扫描

1. IF、IFE伪指令及其条件汇编结构

对于 IF 伪指令而言，当给定的数值表达式的值不为零则认为汇编条件成立，否则认为不成立；IFE 伪指令则与之相反。

【例 9.17】求一元二次方式 $ax^2+bx+c=0$ 的实根，无实根则显示“NO REAL ROOT”。试编写程序。

分析：可以使用分支程序来处理无实根和有实根这两种情况。根据给定的 a、b 及 c 的值判断 $b^2-4ac<0$ 是否成立，成立则执行显示“NO REAL ROOT”的程序段，否则执行计算实根的程序段。采用这种方法时，不管 $b^2-4ac<0$ 是否成立，两个程序段都会被汇编生成对应的目标代码。

考虑到求一元二次方程实根的程序段较大，当无实根时没有必要将这段程序汇编生成目标代码，故可以采用条件汇编结构。根据具体的 a、b 及 c 值，在显示“NO REAL ROOT”的程序段及计算实根的程序段中选择一个进行汇编，这就减少了目标代码；在执行程序时，又避免了对条件 $b^2-4ac<0$ 是否成立的条件判断，减少了程序执行时间。条件汇编结构如下：

```
DISP DB 'NO REAL ROOT', 0AH, 0DH, '$'
        M
IF B*B-4*A*C LT 0
   MOV DX, OFFSET DISP
   MOV AH, 09H
   INT 21H ; 显示“NO REAL ROOT”
ELSE
        M              ;  计算一元二次方程实根的程序段。
ENDIF
   IF 伪指令中的表达式
   B*B-4*A*C LT 0
```

只有两个取值：B*B-4*A*C 小于 0 时表达式取值 1，即汇编条件成立；大于等于 0 时表达式取值 0，即汇编条件不成立。

2. IFDEF、IFNDEF伪指令及其条件汇编结构

对 IFDEF 伪指令而言，当给定的符号已经在本模块中定义或已经在本模块中用 EXTRN 伪指令说明为外部符号，则认为汇编条件成立，否则认为不成立。IFNDEF 伪指令则与之相反。

【例 9.18】以下条件汇编结构根据标号 SYM 是否被定义或说明来确定将过程 SUB1 定义为远过程还是近过程。具体而言，当标号 SYM 已被定义或已用 EXTRN 伪指令说明为外部符号，则将 SUB1 过程定义为远过程；否则将 SUB1 过程定义为近过程。

```
IFDEF SYM
  SUB1 PROC FAR
ELSE
  SUB1 PROC NEAR
ENDIF
        M              ; SUB1 过程体。
SUB1 ENDP
```

3. IFB、IFNB伪指令及其条件汇编结构

该伪指令一般用在宏定义的宏体中，伪指令给定的参数为宏定义的形参。对 IFB 伪指令而言，当宏调用时没有给出该形参对应的实参，则认为汇编条件成立，否则认为不成立；IFNB 伪指令则与之相反。

【例 9.19】试定义一个实现无条件转移或条件转移的宏指令。当宏调用时给出一个实参，

则将其作为无条件转移的目标位置；当宏调用时给出两个实参，则将第一个实参作为条件转移的目标位置，将第二个参数是否为零作为转移的条件——为零则转，非零则不转。

分析：可以将该宏定义的宏体设置为条件汇编结构。程序块 1 实现无条件转移，使用第一个形参作为形式上的转移目标位置；程序块 2 实现条件转移，使用第一个形参作为形式上的转移目标位置，使用第二个参数作为形式上的转移条件。而宏调用时是否给出第二个实参可以作为选择汇编程序块 1 或汇编程序块 2 的条件。宏定义如下：

```
GOTO MACRO DEST, COND
    IFB <COND>
        JMP DEST
    ELSE
        MOV AX, COND
        CMP AX, 0
        JZ DEST
    ENDIF
    ENDM
宏调用
        ⋮
GOTO BR1
        ⋮
GOTO BR2, W_DATA
        ⋮
```

对应的宏扩展为：

```
        ⋮
+ JMP BR1
        ⋮
+ MOV AX, W_DATA
+ CMP AX, 0
+ JZ BR2
        ⋮
```

4. IFIDN、IFDIF伪指令及其条件汇编结构

该伪指令一般用在宏定义的宏体中，伪指令给定的参数之一一般为宏定义的形参，给定的另一个参数作为与此形参的对比者。两者的比较实际上是在形参由实参取代后进行，并且是逐字符对比。对于 IFIDN 伪指令而言，当上述对比的结论是完全相同则认为汇编条件成立，否则认为不成立；IFIDN 伪指令则与之相反。

【例 9.20】定义一个宏指令以实现键盘输入一个字符，并将该字符送宏调用指定的 8 位寄存器或存储单元。

分析：通过 01H 号系统功能调用可以实现从键盘输入一个字符并送 AL 的功能。可以在宏定义时设置一个形参，将 01H 号系统功能调用后得到的（AL）传送到此形参。在宏调用时只要通过实参指定一个 8 位寄存器或存储单元即可。但是，如果宏调用时给定的实参也是

AL，则宏扩展中就会出现指令

```
+ MOV AL, AL
```

可见，这种情况既无谓地增加了目标代码又增加了程序执行时间。为此，可以在宏定义中通过条件汇编伪指令对被实参取代的形参进行判断，若宏调用提供的实参不为 AL，则传送（AL）；否则不作传送。宏定义如下：

```
INPUT MACRO CHAR
        MOV AH, 01H
        INT 21H
    IFDIF <CHAR>, <AL>
        MOV CHAR, AL
    ENDIF
    ENDM
```

宏调用

```
    M
INPUT B_DATA
    M
INPUT AL
    M
```

对应的宏扩展为

```
    M
+ MOV AH, 01H
+ INT 21H
+ MOV B_DATA, AL
    M
+ MOV AH, 01H
+ INT 21H
    M
```

考虑到调用此宏指令时，直接用 AL 存放输入字符的需求较为常见，可以使用 IFNB 伪指令对上述宏定义作如下修改：

```
INPUT MACRO CHAR
        MOV AH, 01H
        INT 21H
    IFNB <CHAR>
    IFDIF <CHAR>, <AL>
        MOV CHAR, AL
    ENDIF
    ENDIF
    ENDM
```

当使用不带实参的宏调用

```
INPUT
```

即可得到宏扩展：

```
+ MOV AH, 01H
+ INT 21H
```

说明：IFIDN 伪指令及 IFDIF 伪指令在对两个字符串作对比时会区分字母的大小写。若希望对比时忽略大小写的区别，只要将两者分别改为 IFIDNI 和 IFDIFI。

5. IF1、IF2 及其条件汇编结构

宏汇编程序在对源程序作汇编时采用两遍扫描。第一遍是宏处理，包括建立一个包含符号名、段名、类型和偏移地址的符号表。第二遍则利用符号表及宏汇编程序的内部表格生成机器代码。IF1 伪指令、IF2 伪指令分别规定给定的程序块被包括在第一、第二次扫描中。若不使用这两条伪指令，则源程序在汇编时被两次扫描。

为了加快汇编速度，可以将源程序中的宏定义仅作第一次扫描，如

```
IF1
EXAMP MACRO
        M
      ENDM
ENDIF
```

说明：

（1）条件汇编伪指令与高级语言中的条件语句在形式上相似，但两者具有本质的不同。条件汇编伪指令不产生目标代码，只是在汇编时根据条件指示汇编程序是否对某一段源程序作汇编。在目标程序运行期间，条件汇编指令则不起作用。

（2）条件汇编伪指令与条件转移指令均具有条件判断功能，但彼此不能取代。原因有二：其一，条件汇编指令是根据条件对某段程序是否被汇编做出选择 ；而条件转移指令是根据条件对某段程序是否被执行作出选择。其二，条件汇编指令所依据的条件来源于汇编的结果；而条件转移指令所依据的条件来源于执行的结果。

9.4 库的使用

具有高级语言编程经验者知道，使用库可以使编程工作更为方便。在进行汇编语言程序设计时也可以通过使用库来节省编辑源程序的时间， 减少重复编辑时可能出现的错误，提高源程序的灵活性。

9.4.1 库的建立

将具有通用价值或经常用到的源程序段编辑成一个文件，这就产生了一个汇编语言源程序库。内容为宏定义的源程序库称为宏库。

由于源程序属于公用的实用文件，而且将作为一个源程序段被用于某个调用该库的源程序，所以库的建立要注意以下方面：

（1）其中的宏定义使用的标号必须用 LOCAL 伪指令说明。

（2）其中的宏定义或子程序必须作现场的保护和恢复。

（3）源程序段要尽量具备通用性。

（4）源程序段中不得使用 END 伪指令，这是因为使用源程序库的源程序本身已设置该伪指令。

（5）附上必要的使用说明。

9.4.2 库的使用

源程序在使用库时，只需使用伪指令

INCLUDE 库文件名

汇编过程中，汇编程序处理到这一伪指令时，就会将该库文件中的源程序段插入到该伪指令的位置，然后接着处理其后的程序段。

说明：

（1）要正确选择 INCLUDE 伪指令的位置。当使用的库为宏库时，宜在源程序的开始位置使用该伪指令，以符合宏指令的先定义后调用原则。

（2）当宏库包含多个宏定义，而调用处又不希望使用其中的某些宏定义时，可以在 INCLUDE 伪指令后用 PURGE 伪指令取消不用的宏定义，以达到节省目标代码所占空间的目的。这种取消仅对本次库的使用产生影响，而不影响宏库。

9.5 模块化程序设计

9.5.1 模块化程序设计概述

1. 模块化程序设计的概念

使用汇编语言程序完成一个大任务时，往往需要采用自顶向下，逐步细化的方法，将大任务分成若干个功能模块，必要时还要将这些功能模块划分成更小的功能模块。根据功能模块编写的汇编语言程序通常被称为程序模块，程序模块以 END 语句作为结束标志；程序模块经汇编生成的目标程序通常称为目标模块；多个目标模块可通过连接程序生成完成整个任务的一个可执行程序，即可执行模块。采用上述方法的程序设计就称为模块化程序设计。

2. 模块化程序设计的优点

（1）整个程序由若干模块构成，结构简单明了，符合大规模生产准则。

（2）单个程序模块功能简单，易于编写和调试。

（3）可以利用已有的常用程序模块。常用程序模块由于被反复使用和验证，其正确性已得到保证，且避免了编制程序的重复劳动。

（4）一旦功能模块划分完毕，多个程序员便可以并行工作，分别独立地编制所分配的程序模块，既有利于分工合作，缩短编程周期，又利于明确个人职责。

（5）程序的改错和功能扩展可局部化。

3. 模块化程序设计的一般规则

（1）程序模块应根据功能划分，且大小适度。过大则通用性差，过小则增加系统开销。

（2）程序模块间应保持逻辑上的独立性。尽可能避免用转移指令在各模块之间跳转；模块间的调用采用子程序调用方式，数据交流采用入口参数、出口参数的传递方式；模块内部尽量使用局部变量。

（3）应将为多个模块公用的程序段设置成独立的模块，以减少在多个模块中重复编写这一程序段。

（4）程序模块应力求具有通用性，以提高其利用率。

（5）程序模块应尽量设计成单入口、单出口形式，以利程序的阅读和调试。

4. 模块化程序设计的步骤

（1）根据整个程序应达到的功能，将其划分为多个不同层次的功能模块，并画出层次图，如图 9.1 所示。值得注意的是，有些功能模块，如显示模块可能作为多个不同层次，不同位置的子模块，如图 9.1 中的模块 C。

（2）确定每个功能模块的任务，确定如何在功能模块中建立段，以及确定各个段的定位方式、组合方式和'类别'，以便与其他相关功能模块中的段适当组合。

（3）确定每个功能模块与其他相关功能模块之间如何通信。

（4）编制各个程序模块，包括给出程序模块的说明；将各个程序模块分别做汇编，生成各自的目标模块和可执行模块并调试。

（5）使用连接程序将各个目标模块连接起来形成一个可执行模块并做整体调试。

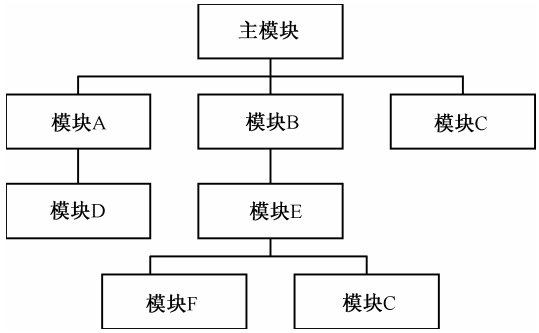


图 9.1　功能模块层次图

9.5.2　段的定义

汇编语言程序采用了段结构，即一个程序模块由若干个段组成。一个完整程序的各个程序模块之间的联系往往离不开各个程序模块的多个段之间的联系。

如上所述，各个程序模块生成的目标模块由连接程序连接成一个可执行模块。连接工作包括对各模块的多个段的组合，而对多个段的组合的一个重要依据就是各个程序模块中段的定义。MASM 5.0 以上版本不仅提供了低版本所具有的完整段定义功能，而且还提供了简化段定义功能。

1. 完整段定义

在第 4 章中曾经论及段定义的格式，在多个程序设计举例中也多次使用了段定义，但涉及的都是段定义的最基本的内容，在进行段定义时还没有考虑到与其他模块中的段之间的联系问题。以下对完整段定义进行具体介绍。

（1）一般格式

段名 SEGMENT ［定位方式］［组合方式］['类别']

　　段体

段名 ENDS

（2）功能

通过 SEGMENT 和 ENDS 伪指令将由多个语句组成的段体定义为一个段，且由 SEGMENT 伪指令中的定位方式、组合方式及 '类别' 对该段做出具体约定，以作为连接程序进行段的组合工作的依据。

段名由用户自定。若希望该段与其他程序模块中的某个段建立组合关系，则应使这两个分属不同程序模块的段使用同一段名。定位方式、组合方式及'类别'均属可选项，彼此用空格相隔，顺序不可改变。以下分别介绍这三个选项的含义。

（1）定位方式

定位方式确定该段起始地址的要求。所有定位方式及其起始地址的要求如表 9.4 所示。

表 9.4　定位方式及起始地址要求

定 位 方 式	段的起始地址	说　　明
BYTE	×××× ×××× ×××× ×××× ××××B	该段从下一个可用字节地址开始
WORD	×××× ×××× ×××× ×××× ×××0B	该段从下一个可用字地址开始
DWORD	×××× ×××× ×××× ×××× ××00B	该段从下一个可用双字地址开始
PARA	×××× ×××× ×××× ×××× 0000B	该段从下一个可用节地址开始，此为默认定位方式
PAGE	×××× ×××× ×××× 0000 0000B	该段从下一个可用页地址开始

存储空间的一节为 16 字节，且每个节地址的低 4 位为 0000B；一页为 256 字节，且每个页地址的低 8 位为 0000 0000B。

【例 9.21】 设有对 A1 段和 A2 段的段定义如下，则所在程序被汇编和连接后，这两个段在存储器中的位置如图 9.2 所示。

```
A1   SEGMENT   PAGE
     DB   23H   DUP   （61H）
A1   ENDS
A2   SEGMENT
     DB   30H   DUP（0）
A2   ENDS
```

由于 A1 段的定位方式为 PAGE，故该段从下一个可用页地址开始，即从××××:××00 开始；A1 段定义了 23H 字节的空间，故该段末地址为××××:××22。

由于 A2 段未指定定位方式，故被默认采用定位方式 PARA，即从下一个可用的节地址开始，即从××××:××30 开始，××××:××23 至××××:××2F 共 13 个字节未使用。

（2）'类别'

'类别'是一个字符串，单引号中的字符可由编程者自定。连接多个模块时，同'类别'的所有段（不管段名是否相同）被放在连续的存储区中，先出现者在前，后出现者在后。

（3）组合方式

在连接多个模块时，若多个模块中定义的段使用了不同的段名和'类别'，将使得总模块中存在多个数据段、堆栈段及代码段，但就 8086/8088 而言，它只允许访问四个段。解决这一矛盾的一个简单办法是，各个模块的数据段、堆栈段及代码段分别取同样的段名和'类别'，以便总模块中仅存在一个数据段、一个堆栈段及一个代码段。仍就 8086/8088 而言，它规定

一个段须小于 64KB。于是，这一简单办法便具有了明显的局限性。在段定义中选用各种组合方式和'类别'可以解决上述问题。各种组合方式如表 9.5 所示。

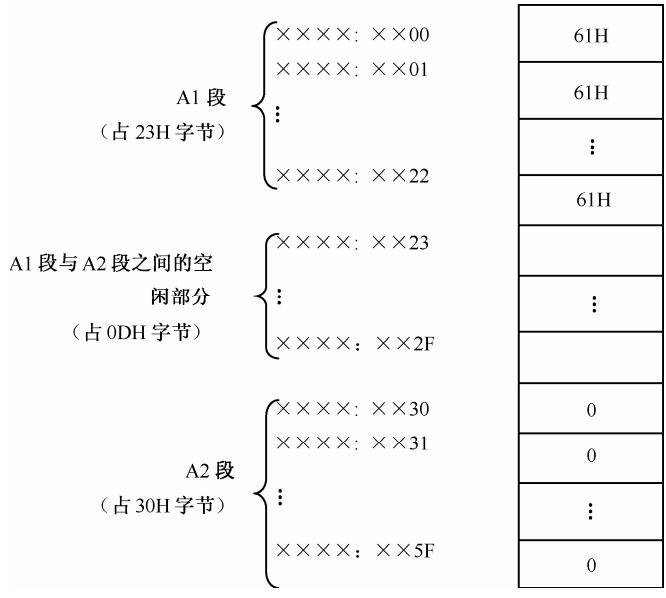


图 9.2　程序中各段定位示意图

表 9.5　组合方式

组 合 方 式	含　　义
PUBLIC	与其他模块中使用 PUBLIC 组合方式的同名同'类别'段合并为一个段
STACK	与 PUBLIC 同，且组合而成的段固定为堆栈段
COMMON	与其他模块中使用 COMMON 组合方式的同名、同'类别'段相覆盖，即使用同一个段首地址。组合成的段长度取决于其中的最长者
AT 表达式	使该段从表达式值所确定的段首地址开始存放
MEMORY	使该段定位在较其他各段而言地址更大的存储位置
（默认）	使该段为独立段

【例 9.22】设有 3 个程序模块 E9_2_1.ASM、E9_2_2.ASM 及 E9_2_3.ASM 现将它们对应的目标模块依次连接，则 3 个模块中的段的组合情况如图 9.3 所示。

```
; 程序 E9_2_1. ASM
NAME EXAMPLE9_2_1
A   SEGMENT    PAGE    PUBLIC
    DB    23H    DUP（61H）
A    ENDS
;
B    SEGMENT      STACK
    DB     20H    DUP（0）
B    ENDS
       N
    END
```

```
; 程序 E9_2_2. ASM
NAME   EXAMPLE9_2_2
A   SEGMENT   PUBLIC
    DB      10H    DUP（41H）
A   ENDS
;
B    SEGMENT   STACK
    DB    25H   DUP（0FFH）
B    ENDS
;
C    SEGMENT   COMMON
    DB 1,2,3,4,5,6
C    ENDS
     M
    END
; 程序 E9_2_3. ASM
NAME   EXAMPLE9_2_3
D     SEGMENT    BYTE PUBLIC
      DW    0FFFFH
D     ENDS
;
C      SEGMENT   COMMON
       DB    10H   DUP（11H）
C      ENDS
      M
     END
```

分析：

（1）本例中各个段的'类别'均为空，可以认为'类别'相同。故各段被放在连续的存储区中。

（2）本例中组合方式为 PUBLIC 的段有模块 E9_2_1.ASM 的段 A，E9_2_2.ASM 的段 A，以及 E9_2_3.ASM 的段 D。前两个段同名、同'类别'，故连接时被合并为一个新的段 A。由于所有模块中只有一个段 D，故该段独立存在，等价于不给出段 D 组合方式 PUBLIC 的情况。

模块 E9_2_1.ASM 的段 A 定位方式为 PAGE，占用 23H 字节，故该段从下一个可用页地址开始，即从××××:××00 开始，到××××:××22 结束；模块 E9_2_2.ASM 的段 A 定位方式为默认的 PARA，占用 10H 个字节，故该段从下一个可用节地址开始，即从××××:××30 开始，到××××:××3F 结束。虽然组合方式 PUBLIC 使两个模块中的段 A 合并为一个段，但由于定位方式的限制，这两个段之间存在 0DH 字节的空闲存储区。

（3）本例中组合方式为 STACK 的段有模块 E9_2_1.ASM 的段 B 和模块 E9_2_2.ASM 的段 B，这两个段同名、同'类别'，故连接时被合并为一个新的段 B，且固定作为堆栈段。这两个段的定位方式为默认的 PARA，而它们下一个可用的地址恰好正是节地址，故不存在空闲存储区。

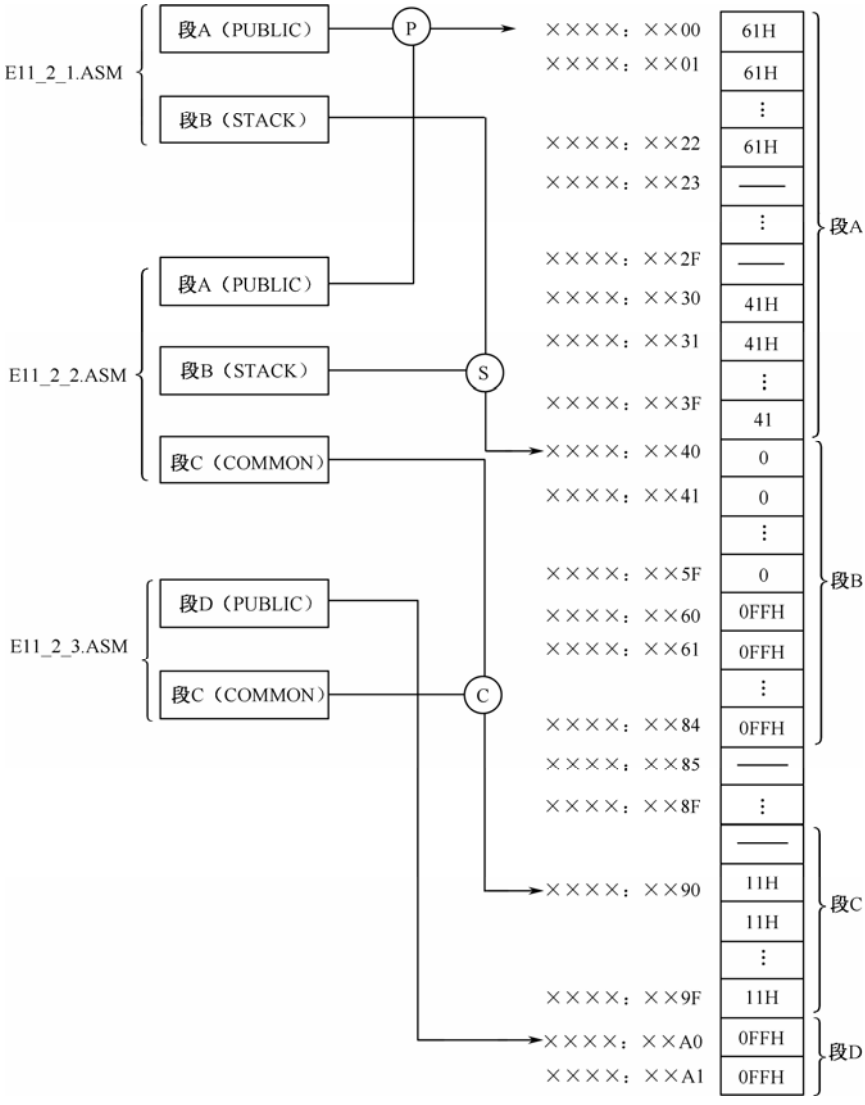


图 9.3　三个模块中各段的组合

（4）本例中组合方式为 COMMON 的段有模块 E9_2_2.ASM 的段 C 和模块 E9_2_3.ASM 的段 C，这两个段同名、同'类别'，故连接时两者相覆盖，形成一个新的段 C。按照题目要求，连接时模块 E9_2_2.ASM 在前，E9_2_3.ASM 在后，故后者的段 C 覆盖前者的段 C。由于定位方式为默认的 PARA，故覆盖后产生的新的段 C 前有 0BH 个字节的空闲存储区。

如果将模块 E9_2_2.ASM 和模块 E9_2_3.ASM 的连接次序交换一下，覆盖后产生的新的段 C 情况如何？请读者思考。

2. 简化段定义

MASM 5.0 以上版本提供了简化段定义的伪指令。使用这些伪指令可以简化段定义，而且有利于汇编语言程序与高级语言程序的连接。

（1）MODEL 伪指令

格式：.MODEL　存储模型

功能：指定汇编程序采用何种存储模型。可采用的存储模型包括：

SMALL 全部数据限制在单个 64KB 的数据段内；全部代码限制在单个 64KB 的代码段内，所有转移均为近转移。

MEDIUM 全部数据限制在单个 64KB 的数据段内；代码可以大于 64KB，可以使用多个代码段，使用的转移可能为近转移，也可能为远转移。

COMPACT 数据量可以大于 64KB；全部代码限制在单个 64KB 的代码段内，所有转移为近转移。

LARGE 数据量、代码量均可大于 64KB；使用的转移可能为近转移，也可能为远转移。

HUGE 数据量、代码量均可大于 64KB；与以上各存储模型不同的是，允许一个数组大于 64KB。

对一般汇编语言程序而言，使用 SMALL 即可满足需要。

（2）DATA 伪指令

格式：.DATA

功能：此为定义数据段伪指令，表示数据段的开始。

（3）STACK 伪指令

格式：.STACK［堆栈字节数］

功能：此为定义堆栈段伪指令，指定椎栈的字节数，默认值为 1024。

（4）CODE 伪指令

格式：.CODE［段名］

功能：此为定义代码段伪指令，表示代码段的开始。

【例 9.23】第 4 章中的示例程序可用简化段定义方式改写如下：

```
NAME    EXAMPLE9_3
.MODEL    SMALL
.DATA
DATA1 DB 4 DUP（1）, 10H, 11, 0AH, 0, 0BH
SUM DB?
COUNT EQU 9
.STACK 100H
.CODE
START:    MOV AX, DGROUP
MOV DS, AX
XOR AL, AL
MOV CX, COUNT
LEA SI, DATA1
LOOP1:        ADD AL, [SI]
              INC SI
              LOOP LOOP1
              MOV SUM,AL
              MOV AH,4CH
              INT 21H
              END START
```

说明：

（1）上述的.DATA、.STACK 及.CODE 伪指令标志段的开始，同时也标志上一个段的结束，如上例中的.CODE 伪指令既表示代码段的开始，又表示堆栈段的结束。整个程序的结束伪指令 END 同时也是最后一个段的结束标志。

（2）在做简化段定义时，编程者一般无须考虑这些段的段名、定位方式、组合方式及'类别'。事实上，MASM 将给以默认值，且自行实现 ASSUME 伪指令功能，将段寄存器设定为这些段的段地址寄存器。在本例中，DGROUP 是 MASM 自行实现 ASSUME 伪指令时，为所有的段设置的段组名。

（3）在一个程序中可以既使用完整段定义，又使用简化段定义。此时就有必要了解 MASM 对简化段定义中段的有关默认值的具体内容，以便与完整段定义中的段实现组合。关于具体默认值，请参阅有关资料。

9.5.3 模块间的通信

一个程序中多个模块之间的联系一方面借助于这些模块中多个段之间的组合，另一方面通过模块间的通信来实现。这里所述的通信的含义是：一个模块引用另一模块中定义的符号常量、变量、标号及过程名等标识符。例如，模块 A 使用模块 B 中定义的符号常量、变量的值作为原始数据；又如，模块 A 的操作结果送模块 B 中定义的变量；再如，模块 A 使用转移指令转至模块 B 的某一标号处，或者调用模块 B 中定义的子程序。

如前所述，各个程序模块需独自汇编生成目标模块后方能由连接程序生成最终的一个可执行模块。在对一个程序模块作汇编时，若程序模块中使用了未在本模块中定义的符号常量、变量、标号及过程名等标识符，则汇编程序就会给出“Symbol not defined”，即“符号未定义”的出错信息。为此，在使用这种符号的模块中有必要采取一定的措施，对这些标识符给出声明，指出这些标识符在本模块外部定义；另一方面，在定义这些标识符的模块中有必要对这些标识符做出声明，指出这些标识符可以为其他模块引用。

1. 模块通信伪指令

以上所述的措施就是，在定义了可为其他模块引用的标识符的模块中使用 PUBLIC 伪指令；在需引用其他模块所定义的标识符的模块中使用 EXTRN 伪指令。

（1）PUBLIC 伪指令

格式：PUBLIC 标识符[，标识符，…]

功能：将列出的标识符声明为公共标识符，以便其他模块引用。

说明：

① 标识符可以是符号常量、变量、标号及过程名。

② 标识符必须在本模块中定义。

③ 该伪指令可以放在本模块的任何位置，为清晰起见，一般放在模块起始处，如紧接在 NAME 伪指令之后。

（2）EXTRN 伪指令

格式：EXTRN 标识符：类型[，标识符：类型，…]

功能：指出列出的标识符在本模块中引用，但是在其他模块中定义，且在定义这些标识符的模块中使用 PUBLIC 声明为公共标识符。另外，还指出各个标识符的类型。

说明：

① 标误符可以是符号常量、变量、标号及过程名。

② 这里指出的标识符的类型必须与定义时的类型一致。

在其他模块使用 EQU 或=伪指令定义为符号常量，则这里的类型为 ABS，意指纯数值。在其他模块使用 DB、DW 及 DD 等伪指令定义为变量，则这里的类型相应为 BYTE、WORD、DWORD 等。在其他模块中作为某个语句的标号，或作为过程定义伪指令中的过程名，则这里的类型为 NEAR 和 FAR 之一。

③ 这些标识符不得在本模块中再做定义。

④ 该伪指令可以放在本模块的任何位置，但一般放在模块起始处。放在起始处有两个优点：其一，相对清晰；其二，避免因在该指令之前引用其他模块所定义的标识符而发生错误。

【例 9.24】以下程序由 3 个模块 E9_4_1.ASM、E9_4_2.ASM 及 E9_4_3. ASM 组成，3 个模块之间的联系一方面借助于它们段之间的组合，另一方面通过模块间的通信来实现。程序功能是从键盘接受 60 个字符，且每接受一个字符就在屏幕上显示。

```
; 程序 E9_4_1.ASM
NAME    EXAMPLE9_4_1
            PUBLIC COUNT        ; 指出本模块定义的标识符 COUNT 可为其他模块引用;
            EXTRN SUB1:FAR      ; 指出本模块将引用其他模块定义的 FAR 型标识符 SUB1;
            COUNT EQU 60
C           SEGMENT PUBLIC 'CODE'
            ASSUME  CS:C
START:      CALL FAR PTR SUB1   ; 调用远过程 SUB1;
            MOV AH,4CH
            INT 21H
C           ENDS
            END START
; 程序 E9_4_2. ASM
NAME        EXAMPLE9_4_2
            PUBLIC SUB1         ; 指出本模块定义的标识符 SUB1 可为其他模块引用;
EXTRN       COUNT:ABS, SUB2:NEAR  ; 指出本模块将引用其他模块定义的符号常量
                                      COUNT, 以及 NEAR 型标识符 SUB2;
C1      SEGMENT  PUBLIC   'CODE'
ASSUME  CS:C1
SUB1    PROC  FAR
        MOV  CX,COUNT
LOOP1:  MOV  AH,8
        INT  21H            ; 从键盘接受字符并送 AL;
        CALL  SUB2          ; 调用近过程 SUB2;
        LOOP  LOOP1
        RET
SUB1    ENDP
```

```
C1          ENDS
            END
; 程序 E9_4_3. ASM
NAME        EXAMPLE9_4_3
            PUBLIC   SUB2      ; 指出本模块定义的标识符 SUB2 可为其他模块引用;
C1          SEGMENT       PUBLIC   'CODE'
            ASSUME   CS:C1
SUB2        PROC   NEAR
            MOV   DL,AL
            MOV   AH,2
            INT   21H             ; 显示字符。
            RET
SUB2        ENDP
C1          ENDS
            END
```

从程序可以看出以下几点：

① 程序采用了第 8 章图 8.5 所示的子程序嵌套的结构，即主程序调用子程序 SUB1，而子程序 SUB1 又调用子程序 SUB2。特别之处在于，主程序、子程序 SUB1 及子程序 SUB2 分属三个模块。

② 三个模块的代码段均为 'CODE' 类别，以便连接后被放在连续的存储区中。其中后两个模块 E9_4_2.ASM 及 E9_4_3.ASM 代码段不仅同'类别'，而且同为 PUBLIC 组合方式，同取段名 C1，故连接后组合成为一个新的代码段 C1。

③ 三个模块通过对其他模块所定义的过程（即子程序）的调用，以及对其他模块段定义的符号常量的调用实现模块间的通信。

首先，第一个模块 E9_4_1.ASM 调用了第二个模块 E9_4_2.ASM 定义的过程 SUB1。由于连接后这两个模块中的代码段不能组合为一个新的代码段，故这种调用应为远调用。由于这一原因，第一个模块 E9_4_1.ASM 必须使用 EXTRN 伪指令，将所要引用的过程名 SUB1 声明为 FAR 型；相应地，第二个模块 E9_4_2.ASM 必须使用 PUBLIC 伪指令，将 SUB1 声明为公共标识符。

其次，第二个模块 E9_4_2.ASM 调用了第三个模块 E9_4_3.ASM 定义的过程 SUB2。由于连接后这两个模块中的代码能够组合成为一个新的代码段，故这种调用是在同一代码段内进行，即为近调用。由于这一原因，第二个模块 E9_4_2.ASM 使用 EXTRN 伪指令，将所要引用的过程名 SUB2 声明为 NEAR 型；相应地，第三个模块 E9_4_3. ASM 使用 PUBLIC 伪指令，将 SUB2 声明为公共标识符。

另外，第二个模块 E9_4_2.ASM 中使用的循环初值引用了第一个模块 E9_4_1.ASM 中定义的符号常量 COUNT，故第二个模块 E9_4_2.ASM 必须使用 EXTRN 伪指令，将所要引用的标识符声明为 ABS 型，相应地，第一个模块 E9_4_1.ASM 必须使用 PUBLIC 伪指令，将 COUNT 声明为公共标识符。

【例 9.25】使用简化段定义，可以将上例中的程序改写如下：

```
; 程序 E9_5_1. ASM
NEME     EXAMPLE_9_5_1
PUBLIC   COUNT
EXTRN    SUB1:NEAR
.MODEL   SMALL
 COUNT   EQU 60
.CODE
START:   CALL   SUB1
         MOV    AX,4CH
         INT    21H
         END    START
; 程序 E9_5_2. ASM
NAME     EXAMPLE9_5_2
PUBLIC   SUB1
EXTRN    COUNT: ABS,SUB2:NEAR
.MODEL   SMALL
.CODE
SUB1     PROC
         MOV    CX,COUNT
LOOP:    MOV    AH,8
         INT    21H
         CALL   SUB2
         LOOP   LOOP1
         RET
SUB1     ENDP
         END
; 程序 E9_5_3. ASM
NAME     EXAMPLE9_5_3
PUBLIC   SUB2
.MODEL   SMALL
.CODE
SUB2     PROC
MOV      DL,AL
MOV      AH,2
INT      21H
RET
SUB2     ENDP
         END
```

在汇编时，MASM 将对以上 3 个模块中的代码段给以相同的默认段名，定位方式，组合方式及'类别'，经过连接就会组合成为同一个代码段。所以，不仅在第二个模块 E9_5_2.ASM 中将 SUB2 声明为 NEAR 型，而且在第一个模块 E9_5_1. ASM 中也将 SUB1 声明为 NEAR 型。

9.5.4 模块的连接

当一个程序的多个程序模块分别被汇编，并生成对应的目标模块后，即可使用连接程序 LINK 将各目标模块连接成为一个可执行模块，如图 9.4 所示。

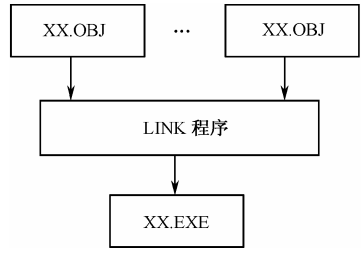


图 9.4 模块连接

【例 9.26】根据例 9.24 中的程序，最终生成对应的可执行文件。

（1）使用 MASM 对程序的三个程序模块作汇编：

```
MASM E9_4_1. ASM
MASM E9_4_2. ASM
MASM E9_4_3. ASM
```

假设使目标模块与程序模块同名，则汇编后生成三个目标模块

```
E9_4_1. OBJ
E9_4_2. OBJ
E9_4_3. OBJ
```

（2）使用连接程序 LINK 将以上 3 个目标模块作连接：

```
LINK E9_4_1+ E9_4_2+ E9_4_3
```

假设给可执行模块取名为 E9_4，则连接后生成一个可执行模块 E9_4. EXE。

说明：

（1）参与连接的各个目标模块所对应的程序模块中，只能有一个程序模块使用伪指令

END 标号

作为程序模块结束标志，其他程序模块使用的伪指令 END 后不能带标号。

（2）在使用连接程序 LINK 实施连接时，要根据具体情况安排各目标模块的次序。这一次序的随意改变将造成可执行模块的改变，有时会因此达不到设计要求。

（3）高级语言程序模块经过编译后也将生成目标模块，使用连接程序可以将其与汇编语言程序模块对应的目标模块连接成一个可执行模块。用这种方法可以实现高级语言程序与汇编语言程序的连接，发挥高级语言使用方便、应用广泛的优点，以及发挥汇编语言效率高、具有直接控制硬件能力的优点。

为了实现这两种目标模块的连接，本汇编语言程序设计中应考虑段的设置，两种语言程序间的调用，以及参数传递等问题。详见有关使用手册。

习　　题

9.1　宏定义可以放在源程序的任何位置，这一结论对不对？

9.2　何谓宏调用、宏扩展和宏嵌套？

9.3　设有以下宏定义

```
FUN MACRO V1, V2
        LOCAL   NEGA, ZERO, OK
        PUSH    AX
        MOV     AL, V1
        CMP     AL, 0
        JS      NEGA
        JZ      ZERO
        MOV     AL, 1
        JMP     OK
NEGA:   MOV     AL, 0FFH
        JMP     OK
ZERO:   MOV     AL, 0
OK:     MOV     V2, AL
        POP     AX
        ENDM
```

若在源程序中进行了两次宏调用

```
        M
FUN B_D1, B_D2
        M
FUN B_D3, B_D4
        M
```

写出后一次宏调用对应的宏扩展。

9.4　定义一个宏指令 CTON，用于将 AL 中的十六进制字符转换为对应的一位十六进制数，定义时要考虑对此宏指令做多次宏调用的情况。

9.5　试用 IRP 及 IRPC 伪指令的重复汇编结构实现【例 9.11】中程序段的功能。

9.6　设源程序中有 6 个近标号，分别为 BR0，BR1，…BR5。试用重复汇编结构定义一个由这 6 个标号构成的跳转表。

9.7　定义一个宏指令 FUNC，用于下列分段函数的计算。所做宏定义要保证宏调用时能根据自变量的不同取值范围做不同的宏扩展，以减少目标代码。

$$y=\begin{cases} x+1 & x<-1 \\ x^2 & -1\leqslant x<4 \\ x-1 & x\geqslant 4 \end{cases}$$

（x 为使用 EQU 伪指令定义的字节有符号数）

9.8　定义一个宏指令，用以对 1～3 个有符号字节变量求取最小值。要求对不同的变量个数产生不同的宏扩展。

9.9　简述模块化程序设计的必要性。

9.10　设有程序模块 EX9_1.ASM 和 EX9_2.ASM，每个程序模块都有一个数据段。若要求在它们对应的目标模块被连接后，这两个数据段中的数据在存储器中连续存放，试做出这两个段的完整定义。

9.11　使用段的简化定义改写例 5.6 中的程序。

9.12　改写例 8.2 中的程序，要求调用程序和子程序分属不同的程序模块。

9.13　说明使用连接程序 LINK 连接多个目标模块时，这些目标模块的先后次序不能随意改变的原因。

第 10 章　系统功能调用及实例

本章在阐述中断概念的基础上，介绍了 DOS（磁盘操作系统）功能调用和 BIOS（基本输入/输出系统）中断调用的基础知识和基本功能。利用这些功能，编程人员不必过问程序的内部细节，就能够非常方便地完成磁盘的读写操作、文件操作、目录操作、内存管理和基本输入/输出管理（键盘、显示等）。

10.1　中断

本节介绍有关中断的基本概念、中断分类以及中断矢量表。

10.1.1　中断的基本概念

中断（Interrupt）是现代计算机输入/输出程序设计常用的控制方式。它是指 CPU 在正常执行程序的过程中，遇到外部/内部的特殊事件需要处理，暂时中断（中止）当前程序的执行，而转去处理特殊事件，待特殊事件处理完后，再返回到暂停处（断点）继续执行原来的例行程序，图 10.1 给出了中断执行路径。

为了能够及时处理这些特殊“事件”，CPU 在中断原有例行程序后，就转入执行事先编写好的专门用于处理这些事件的程序，这些程序叫中断处理子程序或中断服务子程序。原有例行程序被中断的位置叫断点。为了在执行完中断处理程序后能返回到断点处（如图 10.1 中的第 n+1 条指令），CPU 在转入中断处处理程序前，就必须记住“断点”位置以及当时执行例行程序的某些状态（如原有的各寄存器值，标志位置等），这就是保护现场。当然，在执行中断处理程序后，又必须恢复现场，这样才能正确无误地执行原有例行程序。

1. 中断分类

引起中断的原因或者说发出中断请求的来源叫做中断源。根据引发中断的中断源的位置不同，可以把中断分为外部中断和内部中断两类。具体分类如图 10.2 所示。

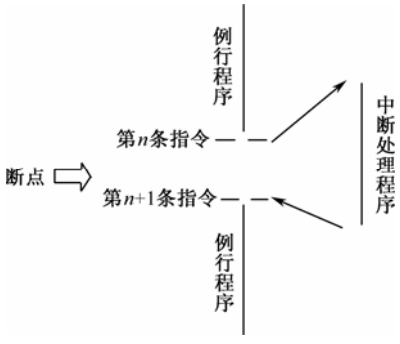


图 10.1　CPU 中断执行路径

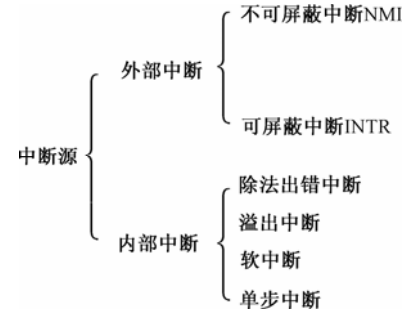


图 10.2　中断源的分类

1）外部中断源

外部中断源来自 CPU 的外部，分以下两种。

（1）不可屏蔽中断源 NMI

不可屏蔽中断源由硬件故障引起，例如电源掉电、存储器出错或者总线奇偶检验错等，这些错误如不及时响应和处理，机器就难以操作下去。

（2）可屏蔽中断源 INTR

可屏蔽中断由各种外设请求中断而产生。当 CPU 处于开中断状态（即状态标志 IF=1）时，能够响应外设的中断请求；当 CPU 处于关中断状态（即状态标志 IF=0）拒绝响应外设的中断请求。因此，称这种中断为可屏蔽中断源。例如，当打印机完成一个字符的打印、键盘完成一个字符输入时，都可以发生中断请求。此时，若 IF=1，则可响应其中断请求；若 IF=0，则不响应其中断请求。在程序允许或禁止响应 INTR 的地方，应分别用 STI 和 CLI 指令将 IF 置 1 或清 0。

2）内部中断源

内部中断源来自 CPU 的内部，其特点是不需要外部硬件支持，不受中断允许标志 IF 的控制。共有 4 种内部中断源。

（1）除法出错，即除数为 0 或商超过了寄存器所能表示的范围时产生。此时，产生一个类型号为 0 的内部中断。

（2）溢出中断指令 INTO

这也是一条软件中断指令，当执行该指令时，若前面的运算已产生溢出，使 OF=1 时，便产生一个内部中断；若 OF=0，便不产生中断。

（3）软件中断指令“INT n”

为了提高编程的效率，减少编程人员对各种硬件设备的了解，系统提供了两组功能子程序：一组在 ROM 的 BIOS（基本输入/输出系统）中，另一组在操作系统 DOS 中。这两组功能子程序实现常用的输入、输出及文件处理基本操作。例如,“INT 21H” 用来调用 DOS 系统功能，每当执行这条指令时，便产生类型号为 21H 的中断，执行事先安排好的中断处理程序。

（4）单步中断

当标志寄存器的标志位 TF 置 1 时，8086/8088 CPU 处于单步工作方式。CPU 每执行完一条指令，自动产生类型为 1 的单步中断，直到将 TF 置 0 为止。使用程序单步执行，为用户提供了强有力的调试手段。例如，在调试程序 DEBUG 中，利用单步中断，每条指令执行后均可查看各寄存器的内容，便以跟踪调试。

为了解决多个中断源同时申请中断时响应的先后顺序问题，系统将所有的中断源划分为 4 级，以 0 级为最高，依次降低，各级情况如下。

0 级——除单步中断以外的内中断源。

1 级——不可屏蔽中断源。

2 级——可屏蔽中断源。

3 级——单步中断源。

不同级别的中断源同时申请中断时，CPU 根据级别高低依次决定服务顺序。

2. 中断矢量表

中断矢量表反映中断类型码与对应的中断处理程序之间的连接关系。该表占用内存中从

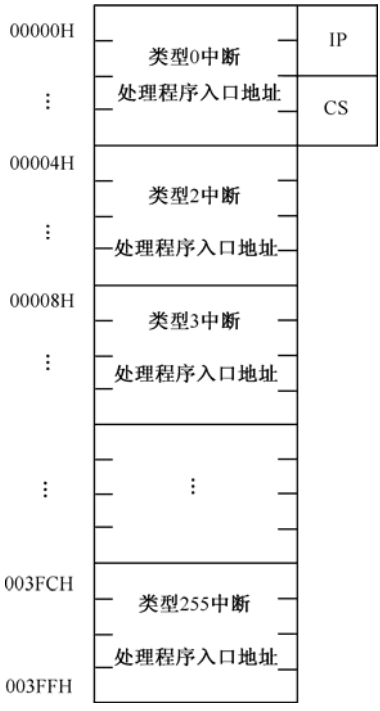


图 10.3　中断矢量表

00000H 到 003FFH 共 1K 个字节的存储空间。表内容分为 256 项，对应中断类型号为 0～255。每项占 4 个字节，用来存放中断处理子程序的入口地址信息，高地址的两字节用来存放中断处理子程序所在段的段首地址，低地址的两个字节用来存放中断处理子程序入口处在段内的偏移地址，其结构图如图 10.3 所示。

每一个中断类型号都有与之对应的中断名，在附录 F 中的表 F.2 列出了常用中断处理子程序及其对应的中断类型号。

根据中断表，CPU 响应中断时即可得到相应中断服务程序的入口地址。例如，打印机的中断类型号为 0FH，其中中断处理程序的入口地址存放在 4*0FH～4*0FH+3 共 4 个字节中。而 DOS 系统功能调用的中断类型号为 21H，其中断处理程序的入口地址存放在 4*21H～4*21H+3 共 4 个字节中。当 CPU 响应 n 号中断源的中断请求时，从 4*n～4*n+3 共 4 个字节中取出 n 号中断处理程序的入口地址送入 IP 和 CS 中，从而使 CPU 执行中断处理程序。

10.1.2　中断的处理过程

一个完整的中断过程，通常要经历中断请求、中断响应、中断处理及中断返回 4 步。

1. 中断请求

这是引起中断的第 1 步，中断源产生中断请求的条件，对不同的中断源是不一样的。如除法出错中断请求的条件是执行除法指令时，除数为零或商过大；输入设备待输入一个字符后提出中断请求；输出设备是在从端口取走一字符后产生中断请求。

2. 中断响应

CPU 在每执行完一条指令后，均要查询是否有中断请求。若有，则按图 10.4 所示的顺序给予响应或不响应。查询中断请求按优先级从高到低的顺序进行。首先查看除法出错、INT n、INTO 指令的内部中断源，然后是 NMI，再后是 INTR，最后是单步中断。除 INTR 之外，其余都是非屏蔽性中断，只要它们发生中断请求，CPU 就响应，且它们的中断类型码是固定的，不需要专门去识别中断类型码。如果是可屏蔽中断 INTR 发出中断请求，CPU 还要根据 IF 的状态来确定是否给予响应。如果 CPU 响应，就识别并读出得到响应的中断源的类型码。当 CPU 对中断请求给予响应后，CPU 立即完成下列操作：

（1）标志寄存器（Flags）压入堆栈；

（2）代码段寄存器（CS）和指令指针（IP）压入堆栈；

（3）IF=0，TF=0。

然后，根据中断类型码从中断矢量表中找到中断服务程序的入口地址，并分别送入 IP 和 CS，接着转入下一步。

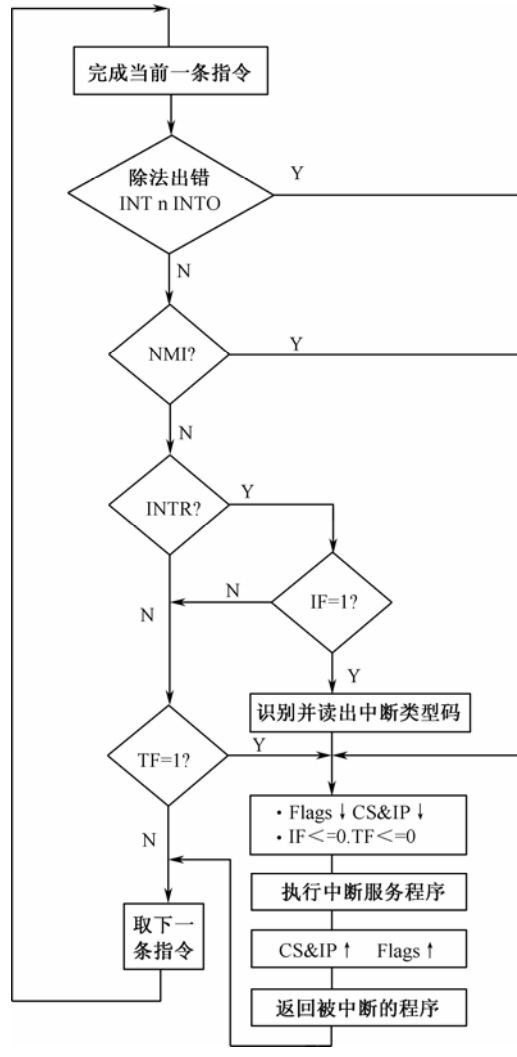


图 10.4　中断处理顺序

3. 中断处理

中断处理是由中断服务子程序来完成的。由于各中断源要求处理的事情不一样，所以各中断处理子程序也就不相同。中断的发生是随机的，不能完全预料在哪个时刻产生中断。中断结束后又要返回到被中断的位置，而中断处理又是执行一段程序来实现的。因此，中断处理子程序的开始处，常常要保护现场，主要是把 CPU 中寄存器（AX、BX、CX、DX、SI、DI、BP、DS、ES）压入堆栈。在中断处理结束时，把堆栈中的这些内容恢复到各寄存器中。保护现场时，需要把哪几个寄存器内容压入堆栈，要根据中断处理程序的具体情况而定。所以中断处理子程序的基本形式如下：

```
INTERRUP  PROC    FAR
          PUSH    AX              ；保护现场
          PUSH    BX
          PUSH    CX
          PUSH    DX
```

```
        PUSH    SI
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
        ┆                       ；中断处理
        ┆
        POP     ES              ；恢复现场
        POP     DS
        POP     BP
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        IRET                    ；中断返回
INTERRUP ENDP
```

4. 中断返回

每一个中断处理子程序的末尾都要使用中断返回指令 IRET。即从堆栈中首先弹出中断断点地址，分别送入 IP 和 CS，然后恢复标志寄存器内容。所以在等待 IRET 指令执行后，便返回到被中断的程序，继续原有程序的执行。

10.2 系统功能调用方法

本节主要介绍汇编语言程序员能控制的软件中断的功能及其使用方法，常用的这类中断有：DOS 功能调用（INT 21H）、BIOS 中断、硬件和外设的中断等。

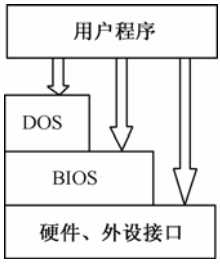


图 10.5 各类系统中断之间的层次关系

图 10.5 给出了程序员可使用的各类中断之间的层次关系。

通常，有些功能既可以通过 DOS 中断调用来完成也可以通过 BIOS 中断调用来完成。例如：打印机输出一个字符的功能，可用 DOS 中断 INT 21H 的功能 5，也可用 BIOS 中断 17H 的功能 0。因为 BIOS 比 DOS 更接近硬件，因此建议尽可能地使用 DOS 功能调用，但在少数情况下必须使用 BIOS 功能，例如：BIOS 中断的 17H 的功能 2 为读打印机状态，DOS 功能调用中没有此功能。

10.2.1 DOS功能调用

DOS 为程序设计人员提供了许多可直接调用的功能子程序。调用时需使用软中断指令：

INT n

其中：n 为中断调用类型号，其值为 00～0FFH。在使用 INT 指令前，应将调用的功能号送入 AH，有关入口参量送入指定寄存器中。若有出口参量，中断调用返回后，其结果在指定的寄存器或存储单元中。DOS 功能子程序的调用使用软中断指令“INT 21H”。该中断有 100 个功能子程序。这些功能子程序可分为以下几种情况。

（1）字符输入和输出

包括键盘输入、显示器与打印机输出和通信口的收发，子功能号为 01H～0CH。

（2）文件管理

分为初级文件管理和高级文件管理两部分。初级文件管理是通过程序段前级 PSP 文件控制块 FCB（File Control Block）来进行的，子功能号为 0DH～24H、27H～29H，这一类文件管理通过建立文件控制块 FCB 进行文件建立、打开、读、写、关闭、删除等操作；高级文件管理直接通过文件代号（文件句柄）进行访问，子功能号为 39H～47H、56H、57H、5AH、5BH，管理功能与初级文件管理基本相同。

（3）内存管理

含分配内存空间、释放内存空间、修改已分配的内存空间等，子功能号为 39H～47H。

（4）作业管理

主要是装入程序、查找匹配文件、中止当前进程和返回调用进程等，子功能号为 48H～4AH。

（5）其他资源管理

如读取和设置中断向量（35H、25H），取日期与时间（2AH～2DH），取国别信息（38H）及设置、读取 Break 检查状态（33H）等。

INT 21H 系统功能调用的方法如下。

（1）送入口参量给指定寄存器

（2）AH ← 功能号

（3）INT 21H

关于字符的输入/输出在本书 5.2.2 部分已有介绍，下面介绍关于文件管理部分。

文件管理分初级文件管理和高级文件管理，功能基本相同。然而，初级文件管理因涉及 FCB 而显得烦琐；而且不支持树形目录结构，不能用于本目录外的文件，也不能在网络上运行。因此，除特殊场合外，目前已很少使用。

高级文件管理支持文件或设备使用“文件号”作为文件标志符。当用户建立或打开一个文件或设备时，在 AX 中返回一个 16 位的二进制数，这就是该文件的文件号。对该文件的文件的读、写和关闭等操作，都必须通过文件号来完成。

高级文件管理仅以文件号为线索，简单易行，且完全适用于单机和网络环境，是目前文件管理的主导方法。本部分将介绍高级文件管理的一些功能。

（1）在当前目录下创建文件

调用：　AH=3CH
　　DX=目录路径名缓冲区偏移地址
　　DS=目录路径名缓冲区所在的段基址
　　CX=文件属性（0—读/写，1—只读，2—隐含）

返回：　文件建立成功时：CF=0，AX=文件代号
　　文件建立失败时：CF=1，AX=错误代码（3—找不到路径，4—文件打开太多，5—不能存取）

（2）打开文件

调用：　AH=3DH

DX=目录路径名缓冲区偏移地址

DS=目录路径名缓冲区所在的段基址

AL=访问属性（0—读，1—写，2—读/写）

返回：　文件打开成功时：CF=0，AX=文件代号

文件打开失败时：CF=1，AX=错误代码（2—未找到指定文件，3—路径不对，5—不能读写，12—调用时 AL 设置错）

（3）关闭当前文件

调用：　AH=3EH

BX=文件代号

返回：　文件关闭成功时：CF=0

文件关闭失败时：CF=1，AX=6

（4）读指定文件

调用：　AH=3FH

BX=文件代号

CX=读入字节数

DX=目录路径名缓冲区偏移地址

DS=目录路径名缓冲区所在的段基址

返回：　文件关闭成功时：CF=0，AX=文件代号

文件关闭失败时：CF=1，AX=错误代码（5—不能读，6—文件控制字无效）

（5）写文件

调用：　AH=40H

BX=文件代号

CX=写盘字节数

DX=目录路径名缓冲区偏移地址

DS=目录路径名缓冲区所在的段基址

返回：　写成功时：CF=0，AX=实际写入的字节数

写失败时：CF=1，AX=错误代码（5—不能写，6—文件控制字无效）

（6）删除一个指定文件

调用：　AH=41H

DX=目录路径名缓冲区偏移地址

DS=目录路径名缓冲区所在的段基址

返回：　删除成功时：CF=0

删除失败时：CF=1，AX=错误代码（2—找不到指定文件，5—指定文件是目录或只读文件，不能删除）

（7）移动文件指针

调用：　AH=42H

BX=文件代号

CX：DX= 移动字节数

AL=移动方式（0—从文件首开始移动，1—从当前位置移动，2—从文件尾反向移动）

返回：　移动成功时：CF=0

移动失败时：CF=1，AX=错误代码

10.2.2　BIOS功能调用

和 DOS 中断服务例程一样，ROM BIOS 服务例程也是以中断服务程序的形式提供的，ROM BIOS 在其初始化期间将所提供的服务例程的地址登记在中断矢量表中，而应用程序想对其调用时发出“INT n”指令即可。因为以中断方式工作，故对 ROM BIOS 的调用也称为 BIOS 中断调用。

1. BIOS 中断

PC 机 1MB 内存空间中最高端（最大地址空间处）是系统 ROM 区，安装在此区中的 ROM 内固化了 ROM BIOS（Basic Input/Output System 基本输入/输出系统）代码，这段代码除完成开机时加电自检、对主要 I/O 接口初始化、引导装入 DOS 等功能外，还提供了大量的例程（子程序）供汇编语言程序员调用。通过调用 BIOS 提供的例程，汇编语言程序员在不必了解复杂的硬件接口细节的情况下，也可以轻松完成对硬件的控制，并且还可使程序简洁清晰，因为只需几条指令便可完成对 BIOS 例程的调用，而直接对硬件编程实现相同功能则需要大段代码。

常用的 ROM BIOS 中断情况见表 10.1。

表 10.1　常用的 BIOS 中断

中断号	功能	中断号	功能
10H	显示器	15H	AT 扩充服务
11H	确认设备调用	16H	键盘服务
12H	取内存容量	17H	打印服务
13H	磁盘 I/O	1AH	时间/日期
14H	串行口通信		

对于硬件的访问控制，一般来说有 3 种方法可以考虑，即 DOS 功能调用，BIOS 功能调用或直接访问硬件。在这 3 种方法中 DOS 功能调用层次最高，实际上用户对 DOS 的功能调用是 DOS 经过处理后再转化为对 BIOS 的功能调用，最终由 BIOS 完成对硬件的访问。若需要的操作有相应的 DOS 功能调用，则尽量通过 DOS 功能调用实现，因为高层调用需要了解的细节少，往往参数也少，而且兼容性好。即只要能启动 DOS，这项服务（功能调用）就保证能用。若通过 BIOS 调用实现，则不同机器上 BIOS 代码提供的服务可能不同，所以兼容性要差些。

BIOS 调用在层次上低于 DOS 功能调用而高于直接访问硬件。因为它比 DOS 更接近硬件，故硬件控制功能更强，而且执行速度更快。但是需要了解的硬细节也更多，兼容性也比 DOS 调用差。当需用到 DOS 功能调用没法实现的功能时，或需要追求较高的执行速度时，就需要使用 BIOS 功能调用。例如：DOS 显示功能调用非常简单，而且执行速度也慢，BIOS 则提供了相当丰富的显示功能调用，而且速度快。故在显示控制方面一般都用 BIOS 功能调用。再如：如果因加密需要将磁盘按非标准格式格式化时，DOS 无此功能调用，而 BIOS 功

能调用却允许这样做，因而也必须通过 BIOS 功能调用来完成。

直接访问硬件层次最低，可以控制实现硬件的全部功能，但是需要了解的硬件细节也最多，而且不同机器间若被访问硬件有差异，可能程序不能通用。当需要的功能 BIOS 中断调用也未提供时，只能直接访问硬件实现。例如：要想编一播放音乐的程序，只能直接访问硬件实现。

2. 常用的BIOS功能调用

BIOS 功能调用比 DOS 功能调用在控制底层方面功能更强大，能完成许多 DOS 功能调用无法完成的功能，下面对其常用功能做简单介绍。

（1）磁盘操作

BIOS 中无磁盘文件的概念（文件是 DOS 一级的概念），它“眼中”的磁盘是扇区和磁道。它的 13H 号功能调用提供了详尽的控制磁盘操作的功能，如可以按道格式化磁盘，而且格式化磁道时可以自由设定每道扇区数及每扇区的字节数。同时，每面格式化的磁道数也可以自由设定。这样，加密软件就可以调用这些功能格式化出非 DOS 标准的磁盘结构，在这样的磁盘中记录的数据在 DOS 下不能读取，当然也无法复制，从而达到防复制的目的，并且，这并不妨碍加密者使用磁盘，因为加密者知道磁盘的结构，可以通过设置适当的参数后调用 BIOS 的扇区读写功能，来读写磁盘中的数据。

13H 号功能调用内含的子功能见表 10.2。

表 10.2　BIOS 13H 号功能调用的子功能

AH	功能	AH	功能
00	复位磁盘	0B	写长扇区
01	取磁盘状态	0C	查找柱面（磁道）
02	读扇区	0D	备用磁盘复位
03	写扇区	10	检测驱动器是否准备好
04	检测扇区	11	复校驱动器
05	格式化磁道	14	控制器内部诊断
08	取当前驱动器参数	15	取磁盘类型
09	初始化双驱动器	16	改变磁盘状态
0A	读长扇区	17	置磁盘类型

下面介绍 13H 号功能调用中最常用的扇区读写功能调用。

① 13H 号功能调用 02H 号子功能：读扇区

入口参数：

AH=02，指明读扇区功能调用

AL 要读扇区数

DL 驱动器代号，0 和 1 代表软盘，80H 和 81H 代表硬盘

DH 所读磁盘磁头号，以软盘来说，只能是 0 和 1

CH 10 位磁道号的低 8 位，CL 寄存器的第 6、第 7 位存放其高 2 位

CL 低 5 位为要读的第一个扇区的扇区号（注意扇区号从 1 开始而非从 0 开始）。高 2 位表示磁道柱面号的高 2 位

ES：BX 指出存放从磁盘所读数据的内存地址

出口参数：

读出数据放在 ES:BX 所指的内存区域中

若产生错误，CF 置 1，AH 内为错误代码

② 13H 号功能调用 03H 号子功能：写扇区

入口参数：

AH=03，指明写扇区功能调用

AL 要写扇区数

DL 驱动器代号，0 和 1 代表软盘，80H 和 81H 代表硬盘

DH 所读磁盘磁头号，以软盘来说，只能是 0 和 1

CH 10 位磁道号的低 8 位，CL 寄存器的第 6、第 7 位存放其高 2 位

CL 低 5 位为要写的第一个扇区的扇区号（注意扇区号从 1 开始而非从 0 开始）。高 2 位表示磁道柱面号的高 2 位

ES：BX 指出写往磁盘所数据的内存地址

出口参数：

除非产生错误，信息都写入磁盘。若产生错误，CF 置 1，AH 内为错误代码。

（2）显示功能

① BIOS 的显示功能

DOS 的显示功能相当简单，而 BIOS 的显示功能则非常丰富。DOS 功能调用中不能显示图形，即使显示字符，也不能控制显示位置，不能设置字符颜色等。因而实际应用软件开发中屏幕操作总是调用 BIOS 来完成，而很少仅靠 DOS 调用完成的情况。BIOS 的显示功能集中在 10H 号中断中。其各个子功能基本情况见表 10.3。

② 显示方式

PC 机的基本显示方式有两种，一种是文本方式，一种是图形方式。

a．文本方式。

在文本方式下，屏幕可显示的最小单位是字符，字符可以设置背景色、前景色和闪烁属性。CGA 显示器有 40×25 字符低分辨率和 80×25 高分辨率两种方式。文本方式下屏幕上的每个字符在显存中占两个字节，一个字节对应 ASCII 码，一个字节对应属性值。由于屏幕上最多有 2000 个字符，只需 4000 个字节的显存即可。因而 32K 的显存被划分成 8 份使用，每份 4K，称为一个显示页。一次只有一个显示页中的数据显示在屏幕上，这页称做当前页。系统中有相应的方法控制切换当前页。

表 10.3 BIOS 的 10H 号功能调用的子功能

AH	功能	AH	功能
00	置显示方式	09	写字符和属性
01	置光标类型	0A	写字符
02	置光标位置	0B	置彩色调色板
03	读光标位置	0C	写点
04	读光笔位置	0D	读点
05	选择当前显示页	0E	以电传方式写字符
06	当前显示页上滚	0F	取当前显示页
07	当前显示页下滚	13	写字符串
08	读字符和属性		

b．图形方式。

在图形方式下，屏幕上可控制的最小单位是点，CGA 将显示屏幕划分为 320×200 个像素的中分辨率和 640×200 个像素的高分辨率方式。在中分辨率方式下，每个像素可以有 4 种不同的颜色显示，背景可以有 16 种颜色。在高分辨率方式下，只能以黑白方式显示。在图形方式下，屏幕有光标但不显示。

本节主要介绍字符方式下的显示，图形方式下的显示将在下节介绍。

③ 字符属性

在文本方式下，字符属性用一个字节表示，属性字节格式如下。

D7：闪烁。

D6～D4：背景色。

D3～D0：前景色。

颜色属性与取值对应关系见表 10.4。

表 10.4　颜色属性与取值

颜色	值（Bin）	颜色	值（Bin）	颜色	值（Bin）	颜色	值（Bin）
黑	0000	红	0100	灰	1000	浅红	1100
蓝	0001	紫	0101	浅蓝	1001	品红	1101
绿	0010	棕	0110	浅绿	1010	黄	1110
青	0011	灰白	0111	浅白	1011	白	1111

④ 光标与窗口

在文本方式下，光标可由程序设置其形状或是否显示，由 10H 号功能调用的 1 号子功能完成。也可以调置或读取其位置，由 10H 号功能调用的 2、3 号子功能完成。另外，当前光标处显示的字符也可以读出，由 10H 号功能调用的 8 号子功能完成。

窗口是屏幕上的一个矩形区域，可以独立进行显示文本的输出及上卷和下卷等操作。屏幕上可以同时设置多个窗口。窗口上卷时，超过顶部的行将自动丢失，出现在底部的新行被填为空格。上、下滚动窗口的操作分别由 10H 号功能调用的 6、7 号子功能完成，清屏操作也是借助于这两种操作完成的，只要在其入口参数中设置 AL=0 即可。

⑤ 文本显示

10H 号功能调用中常用的文本显示子功能见表 10.5。

表 10.5　BIOS 的文本显示功能

AH	功能	入口参数	出口参数
1	置光标类型	(CH)0～3=光标开始行 (CL)0～3=光标结束行	
2	置光标位置	BH=页号 DH=行 DL=列	
3	读光标位置	BH=页号	CH=光标开始行 CL=光标结束行 DH=行 DL=列

续表

AH	功能	入口参数	出口参数
6	当前显示页上卷	AL=上卷行数，AL=0 全屏幕为空白 BH=卷入行属性 CH=左上角行号 CL=左上角列号 DH=右下角行号 DL=右下均列号	
7	当前显示页下卷	AL=下卷行数，AL=0 全屏幕为空白 BH=卷入行属性 CH=左上角行号 CL=左上角列号 DH=右下角行号 DL=右下角列号	
8	读光标位置的属性和字符	BH=显示页	AH=属性 AL=字符
9	在光标位置显示字符及其属性	BH=显示页 AL=字符 BL=属性 CX=字符重复次数	
A	在光标位置只显示字符	BH=显示页 AL=字符 CX=字符重复次数	
E	以电传方式写字符	AL=字符 ASCII 码 BL=前景色	

10H 号功能调用的子功能 09H 和子功能 0AH 都能把一个字符传送到显示屏幕，然后光标返回到它的初始位置，所以在当前光标位置上写一个字符之后，必须用 10H 号功能调用的子功能 02H 移动光标到下一个字符位置上。这两种子功能的区别是：AH＝09H 的功能把字符及其属性输出到当前光标位置上，而 AH＝OAH 的子功能只输出字符，它的属性值就是这一位置上先前已具有的属性。OAH 功能在使用黑白显示器时特别方便。如果想在显示字符后自动后移光标位置，则应用 0EH 号子功能。

（3）图形显示程序设计

编制图形程序是程序设计中非常有趣和有价值的工作之一。在图形领域中，汇编语言具有潜在的优点，因为屏幕上显示的一个图像由成千上万个元素组成，处理这些图像需要大量的指令。以速度而论，汇编语言远比高级语言快得多，最高级的图形技术，例如动画，往往以汇编语言编写才更逼真、有效。

① 显示功能

如上节所述，BIOS 的显示功能全部集中在 10H 号功能调用中。10H 号功能调用中常用的图形方式子功能见表 10.6。

② 图形方式下的分辨率与颜色

显示器（实际由显示卡控制）默认的显示方式是文本方式，要想设置为图形方式，需要调用 INT 10H 的 00H 号子功能完成。

在图形方式中，屏幕被视为 M×N 的点阵，每个点的坐标上的图像元素就是一个像素。在图形方式下显存中要对屏幕上的内容按点存储，由于显存容量有限，在 320×200 中分辨率下，每个点（像素）的颜色用 2 个二进制位来表示，故可有 4 种颜色；在 640×200 像素的高分辨率下，每个点（像素）的颜色用一个二进制位来表示，故只有黑白 2 种颜色。

表 10.6　BIOS 的图形显示功能

AH	功能	入 口 参 数	出口参数
0	设置显示方式	AL=00 40×25 黑白文本方式 AL=01 40×25 彩色文本方式 AL=02 80×25 黑白文本方式 AL=03 80×25 彩色文本方式 AL=04 320×200 彩色图形方式 AL=05 320×200 黑白图形方式 AL=06 640×200 黑白图形方式	
B	置彩色调色板	BH 指明要执行的功能 =0：设置当前调色板背景色 =1：选择调色板 BL 其含义取决于 BH 值 若 BH=0：此寄存器中设置背景色代码（0～0FH） 若 BH=1：此寄存器中放置调色板号	
C	写点	DX=行（0～199） CX=列（0～639） AL=像素值（中分辨率方式下合法像素值为 0～3，高分辨率方式下为 0 或 1。若 AL 的最高位为 1，表示新的像素与当前屏幕上的像素进行异或显示。）	
D	读点	DX=行（0～199） CX=列（0～639）	AL=像素值

10.3　系统功能调用应用实例

【例 10.1】中断应用实例：编写一个中断处理程序，要求在主程序运行过程中，每隔 10 秒响铃一次，同时在屏幕右上角开窗口显示信息“The bell is ring!”以及时、分、秒。

```
NAME        EXAMPLE10_1
DSEG        SEGMENT
STRING      DB ' The bell is ring!'
LEN         EQU $-STRING
DELAYTIME EQU 182
DSEG        ENDS
SSEG        SEGMENT STACK
SSEG        ENDS
CSEG        SEGMENT
            ASSUME CS:CSEG,SS:SSEG,DS:DSEG
            START: MOV AX,DSEG
```

```
        MOV DS,AX

        MOV CX,5                    ;循环次数
        SHOW: PUSH CX
        CALL DELAY                  ;调用子程序 DELAY
        CALL WINDOW                 ;调用子程序 WINDOW

        MOV DL,7                    ;响铃
        MOV AH,2
        INT 21H
        POP CX
        LOOP SHOW

        MOV AH,4CH                  ;返回 DOS 状态
        INT 21H

        ;子程序名称:DELAY
        ;功能:延时
        ;入口参数:无
        ;出口参数:无
DELAY   PROC
        PUSH AX                 ;保存现场
        PUSH BX
        PUSH CX
        PUSH DX

        MOV AH,0                    ;BIOS 中断调用,读取当前时间
        INT 1AH
        MOV BX,DX
        ADD BX,DELAYTIME            ;延时值
        COMP: INT 1AH
        CMP DX,BX                   ;等待
        JNE COMP

        POP DX                      ;恢复现场
        POP BX
        POP CX
        POP AX
        RET
DELAY   ENDP
```

```
;子程序名称:WINDOW
;功能:清屏,开窗口,显示字符串和时间
;入口参数:无
;出口参数:无
WINDOW      PROC
            PUSH AX                     ;保存现场
            PUSH BX
            PUSH CX
            PUSH DX
            PUSH SI

            MOV AH,6                    ;清屏
            MOV AL,0
            MOV CH,0
            MOV CL,0
            MOV DH,26
            MOV DL,79
            MOV BH,7
            INT  10H

            MOV     AH,6                ;开窗口
            MOV     AL,0
            MOV     CH,1
            MOV     CL,59
            MOV     DH,4
            MOV     DL,77

            MOV     BH,0CEH
            INT     10H

            MOV     AH,2                ;设置光标位置
            MOV     BH,0
            MOV     DH,2
            MOV     DL,60
            INT     10H

            MOV     CX,LEN              ;字符串长度
            MOV     SI,0
LP:         MOV     DL,STRING[SI]       ;显示字符串 STRING
```

```
                MOV     AH,2
                INT     21H
                INC     SI
                LOOP    LP

                MOV     AH,2                    ;设置光标位置
                MOV     BH,0
                MOV     DH,3
                MOV     DL,64
                INT     10H

                CALL TIME                       ;调用子程序 TIME 显示
                                                ;系统当前时间

                MOV     AH,2                    ;设置光标位置
                MOV     BH,0
                MOV     DH,25
                MOV     DL,0
                INT     10H

                POP     SI
                POP     DX                      ;恢复现场
                POP     CX
                POP     BX
                POP     AX
        RET
WINDOW          ENDP

;子程序名称:TIME
;功能:显示系统当前时间
;入口参数:无
;出口参数:无
TIME            PROC
                PUSH    AX                      ;保存现场
                PUSH    CX
                PUSH    DX

                MOV     AH,2CH                  ;DOS 中断调用读取系统
                                                ;当前时间(可用 BIOS
                                                ;中断调用)
```

```
                INT  21H

                MOV     AL,CH                   ;显示时
                CALL    BTOD
                MOV     DL,3AH
                MOV     AH,2
                INT     21H

                MOV     AL,CL                   ;显示分钟
                CALL    BTOD
                MOV     DL,3AH
                MOV     AH,2
                INT     21H

                MOV     AL,DH                   ;显示秒
                CALL    BTOD

                POP     DX                      ;恢复现场
                POP     CX
                POP     AX
                RET
TIME            ENDP

;子程序名称:BTOD
;功能:时间由十六进制转十进制
;入口参数:时,分钟,秒分别给 AL
;出口参数:无
BTOD            PROC
                PUSH    AX                      ;保存现场
                PUSH    BX
                PUSH    DX

                XOR     AH,AH
                MOV     BL,10
                DIV     BL
                MOV     BL,AH                   ;保存余数
                MOV     DL,AL
                OR      DL,30H                  ;十位数转为 ASCII 码
                MOV     AH,2
                INT     21H
```

```
                MOV     DL,BL
                OR      DL,30H              ;个位数转为 ASCII 码
                MOV     AH,2
                INT     21H

                POP     DX                  ;恢复现场
                POP     BX
                POP     AX
                RET
        BTOD    ENDP
CSEG  ENDS
                END START
```

【例 10.2】DOS 功能调用实例：编写一程序，在磁盘上创建一个文件，并向其中写入一个字符串。

```
NAME            EXAMPLE10_2
DSEG            SEGMENT
FILE            DB      'c:\masm1.txt',0    ;创建文件的文件名
BUF             DB      'this is a test!'   ;写入文件的内容
ERROR_MESSAGE   DB      0ah,'error!','$'    ;出错后的提示
MESSAGE         DB      0ah,'ok!','$'       ;文件操作成功后的提示
HANDLE          DW      ?                   ;保存文件号
DSEG            ENDS
CSEG            SEGMENT
                ASSUME  CS:CSEG, DS:DSEG
START:
                MOV     AX,DSEG
                MOV     DS,AX
                MOV     DX,OFFSET  FILE
                MOV     CX,0
                MOV     AH,3CH
                INT     21H                 ;创建文件，若磁盘上有此
                                            ;文件，则覆盖
                JC      ERROR               ;创建出错，转 ERROR 处
                MOV     HANDLE,AX           ;保存文件号
                MOV     BX,AX
                MOV     CX,16
                MOV     DX,OFFSET  BUF
                MOV     AH,40H
                INT     21H                 ;向文件中写入 16 个字节内容
                JC      ERROR               ;写出错时，转向 ERROR 处
```

```
                MOV     BX,HANDLE
                MOV     AH,3EH
                INT     21H                 ;关闭文件
                JC      ERROR               ;关闭文件出错，
                                            ;转向 ERROR 处
                MOV     DX,OFFSET MESSAGE
                MOV     AH,9
                INT     21H                 ;操作成功后提示
                JMP     END1
ERROR:
                MOV     DX,OFFSET ERROR_MESSAGE
                MOV     AH,9
                INT     21H                 ;显示错误信息
ENDL1:
                MOV     AH,4CH
                INT     21H
CODE            ENDS
                END     START
```

【例 10.3】BIOS 功能调用实例：编写一程序，读取 A 驱 0 面 10 道 1 扇区中的数据到内存 buf 处，然后将这些数据逐个字节加 1，最后再写回原处。

```
NAME EXAMPLE10_3
DSEG        SEGMENT
ERROR_MESSAGE DB 0AH , 'ERROR !' , '$'          ;出错时的提示
OK_MESSAGE DB 0AH , 'OK !' , '$'                ;成功后的提示
BUF DB 512 DUP(0)                               ;磁盘数据缓冲区
DSEG ENDS
CSEG  SEGMENT
        ASSUME CS: CSEG , DS: DSEG
START:
        MOV     AX , DATA
        MOV     DS , AX
        MOV     AH , 2                          ;读扇区
        MOV     AL , 1                          ;读 1 个扇区
        MOV     DL , 0                          ;读 A 驱
        MOV     DH , 0                          ;读 0 面
        MOV     CH , 10                         ;读第 10 磁道
        MOV     CL , 1                          ;读第 1 扇区
        PUSH    DS
        POP     ES
        MOV     BX , OFFSET BUF                 ;读出数据放到 BUF 处
```

```
        INT     13H                         ;读调用
        JC      ERROR                       ;出错,转到 ERROR 处
        MOV     CX , 512
        MOV     SI , OFFSET BUF
L1:
        INC     BYTE PTR [SI]
        INC     SI
        LOOP L1                             ;这个循环将 BUF 内的 512 个字节
                                            ;各自加 1
        MOV     AH , 3                      ;写扇区
        MOV     AL , 1
        MOV     DL , 0
        MOV     DH , 0
        MOV     CH , 10
        MOV     CL , 1
        PUSH    DS
        POP     ES
        MOV     BX , OFFSET BUF
        INT     13H                         ;写调用,写的位置同上面读的位置
        JC ERROR
        MOV     AH , 9                      ;若都成功,显示成功提示
        MOV     DX , OFFSET OK_MESSAGE
        INT     21H
        JMP     END1
ERROR:                                      ;失败时显示失败提示
        MOV     DX , OFFSET ERROR_MESSAGE
        MOV     AH , 9
        INT     21H
END1:
        MOV     AH , 4CH
        INT     21H
CSEG  ENDS
        END START
```

【例 10.4】BIOS 功能调用实例：编写一程序，首先清屏，然后在 20 行 25 列处显示 10 个星号“*”。

```
NAME EXAMPLE10_4
CSEG  SEGMENT
      ASSUME CS: CSEG
START:
        MOVAH , 6                           ;上滚窗口
```

```
        MOV     AL , 0                  ;上滚行数
        MOV     CH , 0                  ;窗口左上角行号
        MOV     CL , 0                  ;窗口左上角列号
        MOV     DH , 24                 ;窗口右下角行号
        MOV     DL , 79                 ;窗口右下角列号
        MOV     BH , 1FH                ;窗口底部空白行属性：蓝底白字
        INT     10H
        MOV     AH , 2                  ;置光标位置
        MOV     BH , 0                  ;页号 0
        MOV     DH , 20                 ;行号为 20
        MOV     DL , 25                 ;列号为 25
        INT     10H
        MOV     AH , 9                  ;显示字符功能
        MOV     AL , '*'                ;显示字符 ASCII 码
        MOV     BH , 0                  ;设置页号
        MOV     BL , 7                  ;字符属性
        MOV     CX , 10                 ;重复次数
        INT     10H
        MOV     AH , 4CH
        INT     21H                     ;返回 DOS
CSEG  ENDS
        END START
```

习　题

10.1　在 PC 机中中断源有哪些？CPU 响应中断的过程是怎样的？

10.2　DOS 功能调用的用途是什么？与 BIOS 中断调用在使用上有什么差别？

10.3　编一程序，从键盘上读取字符，然后将这一字符按字母顺序的下一个字符显示在屏幕上（如用户输入为 A，则显示 B），共循环接收 20 个。

10.4　编一程序，在 C 盘根目录下创建一个文件 ABC.TXT，然后向其中写入 26 个大写字母作为其内容。

10.5　编程在屏幕上以（10，10）和（50，50）为左上角、右下角顶点，画一红色矩形。

第 11 章　汇编语言与C/C++混合编程及实例

高、低级语言之间的混合编程是开发大型软件中常用的一种技术。汇编语言具有占用存储空间小、运行速度快、有直接控制硬件的能力等特点。但汇编也有它的缺点和不足，例如编写及调试汇编语言程序比高级语言程序要困难、复杂一些，这使得编程者利用它在开发大型软件方面感到“力不从心”，另外不同系列的机器有不同的汇编语言，不同的机器之间不能通用，移植性差；对汇编语言程序员要求较高，需要熟悉机器的内部结构，如存储器结构、寄存器结构，等等。

高级语言是面向用户的语言，具有功能多、表达能力强、使用灵活、应用范围广泛、移植性好等优点，且高级语言程序员不必熟悉计算机内部的具体构造和机器指令，可以把主要精力集中在算法描述上面。但高级语言也存在着代码执行效率低，占用计算机资源多等缺点。

由此可见，每种计算机语言都有自己的优势和劣势。通常在软件开发过程中，大部分程序采用高级语言编写，以提高程序的开发效率。但在某些部分，例如：程序的关键部分、或是运行次数很多的部分、或是运行速度要求很高的部分、或是直接访问硬件的部分等，可利用汇编语言编写，以提高程序的运行效率。所以在汇编语言与高级语言之间常常需要通过彼此联系、取长补短，力图充分利用系统和硬件技术所给予的支持。这种组合多种程序设计语言，通过相互调用、参数传递、共享数据结构和数据信息开发程序的过程就是混合编程。

本章第一节主要介绍汇编语言与 Turbo C 2.0 语言的 16 位混合编程，重点是 C 语言程序调用汇编语言子程序的方法，讨论嵌入汇编方法和汇编语言调用 C 函数的方法。然后论述 Visual C++ 6.0 与汇编语言的 32 位混合编程方法。最后给出几个汇编语言与 C/C++的混合编程的实例。

11.1　Turbo C嵌入汇编方式

C 语言提供了 3 种调用汇编语言功能的方法：嵌入汇编，寄存器伪变量与 bdos()、intdos()、int86()等系列 DOS、BIOS 服务调用函数，C 语言与汇编语言的混合编程。其中第二种方法寄存器伪变量与 bdos()、intdos()、int86()等系列 DOS、BIOS 服务调用函数因主要涉及 C 语言的编程问题，且不同的 C 编译器处理方法不同，本书不做讨论，请读者参考相关 C 语言书籍。

11.1.1　嵌入汇编语句的格式

为了提高 C 语言内某些特殊功能段的处理效率，可以在其源程序中嵌入一段汇编语言程序段，这样做虽然能达到提高程序处理效率的目的，但它也丧失了源程序的可移植性，所以当想用 C 语言和汇编语言混合编程时，程序员需要权衡采用这种方法的利弊。

在 C 语言中嵌入汇编语句的格式如下：

asm [<标号>] <汇编指令或伪指令> <参数> <；或新行>

说明：

asm 是嵌入汇编语句的关键字。

<标号>是可选的。

<汇编指令或伪指令>可以是任何有效的汇编指令或伪指令。

<参数>是汇编指令或伪指令的操作数，它可引用C 语言中的常量、变量或标号。

<；或新行> 都表示asm 语句的结束，一行内可写多条嵌入式汇编语句，它们以“；”分隔。一行内如果只有一个asm 语句，则后面不需要“；”，asm 语句是C 语言中唯一的依靠换行的语句。

asm语句如需要注释，必须采用C 格式的注释，即“/* 注释内容 */”

例如：

```
ASM   MOV AX,DS;                          /* AX ← DS*/
ASM   POP AX; ASM POP DS;ASM RET;         /*合法语句*/
ASM PUSH DS                               /*ASM 语句是 C 程序中唯一可以用换行结*/
                                          /*尾的语句*/
```

编译后，函数外的汇编语句放在数据段内，函数内的汇编语句放在代码段内。

11.1.2 汇编语句访问C语言的数据

内嵌的汇编语句除可以使用指令允许的立即数、寄存器名外，还可以使用 C 语言程序中的任何符号（标识符），包括变量、常量、标号、函数名、寄存器变量、函数参数等；C 编译程序自动将它们转换成相应汇编语言指令的操作数，并在标识符名前加下画线。一般来说，只要汇编语句能够使用存储器操作数（地址操作数），就可以采用一个 C 语言程序中的符号；同样，只要汇编语句可以用寄存器作为合法的操作数，就可以使用一个寄存器变量。

对于具有内嵌汇编语句的 C 程序，C 编译器要调用汇编程序进行汇编。汇编程序在分析一条嵌入式汇编指令的操作数时，若遇到了一个标识符，它将在 C 程序的符号表中搜索该标识符，但 8086 寄存器名不在搜索范围之内，而且大小写形式的寄存器名都可以使用。

【例 11.1】用嵌入汇编方式实现取两数较小值的函数 min。

```
int min(int a,int b)                        /* 用嵌入汇编语句实现的求较小值 */
{
        asm mov ax,a
        asm cmp ax,b
        asm jle minexit
        asm mov ax,b
        minexit:        return(_AX);        /* 将寄存器 AX 的内容作为函数的返回值 */
}
main()                                      /* C 语言主程序 */
{
        min(100,200);
}
```

在 C 语言中使用嵌入式汇编语句时，还应该注意以下几个问题。

1. 通用寄存器的使用

Turbo C 语言中可以直接使用通用寄存器和段寄存器，只要在寄存器名前加一个下画线

就可以了。另外，C 语言中使用 SI 和 DI 指针寄存器作为寄存器变量，利用 AX 和 DX 传递返回参数。

如果 C 语言函数中没有寄存器说明，嵌入式汇编语句可以自动地把 SI、DI 用做暂存寄存器；如果 C 语言函数有寄存器说明，嵌入式汇编语句仍可以使用 SI、DI，但最好采用 C 语言寄存器变量名形式。嵌入式汇编语句可以任意使用 AX/BX/CX/DX 寄存器，以及它们的 8 位形式。

2. 转移指令的标号

内嵌汇编指令可以使用有条件、无条件转移汇编指令和循环指令。但它们只能在函数体内有效，不允许进行段间转移。由于 asm 语句中不能定义标号，因此转移指令的目标必须是 C 语言程序的标号。若该标号很远，转移指令也不会自动地转换成一个远程转移。如：

```
int                 testa()
{
        int   a;
              …
        a:
              …
        asm         jmp   a
              …
}
```

当执行第一内嵌汇编语句时，程序将转到 a：标号指向的语句去执行。转移目标也可以是间接转移，如：

```
int                 testb()
{
        int   a;
              …
a:
              …
        asm         jmp   [a]
              …
}
```

当执行到第一个内嵌式汇编语句时，程序将无条件转移到由 a 整型变量的值作为地址的目标去。

3. C语言结构的引用

嵌入式汇编语句的操作数也可以是 C 语言结构中的某个成员，引用方法采用：

结构变量名.结构成员名

另一种引用的方法是把结构变量的首地址送往某一地址寄存器，然后用该寄存器名（加方括号）再加成员名，中间用圆点隔开。例如：

```
struct      grade{
            int   physics;                  /*物理成绩*/
            int   chinses;                  /*语文成绩*/
            int   english;                  /*英语成绩*/
      } g1;
average()
{
  …
      asm     mov ax,g1. physics        /*取结构体变量 g1 的成员变量 physics*/
      asm     mov di,offset g1          /*取结构体变量 g1 的主存地址*/
      asm     mov bx,[di]. Chinses      /*取结构体变量 g1 的成员 chinses*/
      …
}
```

11.1.3 嵌入汇编的编译过程

C 语言程序中含有嵌入式汇编语言语句时，C 编译器首先将 C 代码的源程序（.c）编译成汇编语言源文件（.asm），然后激活汇编程序 Turbo Assembler 将产生的汇编语言源文件编译成目标文件（.obj），最后激活 TLINK.EXE 将目标文件链接成可执行文件（.exe）。

Turbo C 中的汇编程序默认采用 TASM.EXE。它的格式基本上与微软宏汇编程序 MASM 相同，但是有一些差别。如果采用微软宏汇编程序 MASM，则要使用 3.0 或以后的版本到 5.1 及以前的版本并应将名称改为 TASM.EXE，或者在命令行加上“-E”的 MASM.EXE 选项。由于 Turbo C 2.0 是早期开发的软件，只考虑了与早期 MASM 版本的兼容，而不能与 MASM6.x 很好地配合使用。

Turbo C 2.0 在编译含内嵌汇编语句的程序时，只能采用命令行 TCC 方式，并且要使用“-B”命令行选项编译连接。如果没有加“-B”选项，则在遇到行内汇编时，会报一个警告信息。如果在 C 源程序中使用了#pragma inline 预处理，可以不用命令行选项“-B”。

【例 11.2】将字符串中的小写字母转变为大写字母显示。

```
/*NAME EXAMPLE11_2.c*/
#include<stdio.h>
Void   upper(char *dest,char  *src)
{
            asm     mov     si,src      /* dest 和 src 是地址指针 */
            asm     mov     di,dest
            asm     cld
loop:       asm  lodsb                  /* C 语言定义的标号 */
            asm     cmp     al, ‘a’
            asm     jb      copy        /* 转移到 C 的标号 */
            asm     cmp     al, ‘z’
            asm     ja      copy        /* 不是 a 到 z 之间的字符原样复制 */
            asm     sub     al,21h
```

```
copy:       asm  stosb
            asm      and      al,al
            asm      jnz      loop
}
main()                                          /*主程序*/
{
        char      str[]="This Started Out As Lowercase!";
        char      chr[100];
        upper(chr,str);
        printf("Origin String:\n%s\n",str);
        printf("Uppercase String:\n%s\n",chr);
}
```

编辑完成后，在命令行输入如下编译命令：

TCC –B –Iinclude –Llib EXAMPLE11_2.c

其中：选项“-I”和“-L”分别指定头文件和库函数的所在目录，EXAMPLE11_2.c 是上述编辑的文件。生成可执行文件 EXAMPLE11_2.exe。

程序运行后输出的结果是：

Origin String:

This Started Out As Lowercase!

Uppercase String:

THIS STARTED OUT AS LOWERCASE!

由上例可以看出，内嵌汇编方式把插入的汇编语言语句作为 C 语言的组成部分，不使用完全独立的汇编模块，所以比调用汇编子程序更方便、快捷，并且在大存储模式、小存储模式下都能正常编译通过。

11.1.4 Turbo C模块连接方式

模块连接方式是不同程序设计语言之间混合编程经常使用的方法。各种语言的源程序可以分别编写，并利用各自的开发环境编译形成.OBJ 模块文件，然后将它们连接在一起，生成可执行文件。但是，为了保证各种语言的模块文件的正确连接，必须对它们的接口、参数传递、返回值及寄存器的使用、变量的引用等做出约定，以保证连接程序能得到必要的信息。

本节以 C 语言程序对汇编语言子程序的调用为例说明混合编程的方法，最后也给出了汇编语言程序对 C 语言函数调用的方法。

1. 混合编程的约定

为了能正确连接，在编写 C 语言程序和汇编语言程序时，必须遵循一些共同的约定规则。主要有：命名约定、声明约定、寄存器的使用约定、存储模式约定以及参数传递约定。

（1）命名约定

C 语言编译系统在编译 C 语言源程序时，要在其中的变量名、过程名、函数名等标识符前面加下画线“_”；例如 C 语言源程序中的变量 var，编译后变为_var。而汇编程序在汇编过程中，直接使用标识符，所以在 C 语言程序调用的汇编源程序中，所有的标识符前都要加下

画线“_”。例如汇编语言子程序名为_average()，则汇编后其名字不变。在 C 调用时，可直接使用 average()。这是因为 C 程序编译后，自动将 average()变为了_avarage()，使二者标识符一致。

此外，C 语言对标识的要求是取前 8 个字符有效（Turbo C 在计算机上可以是 32 个字符），并且区分大小写。而汇编语言接收前 31 个字符为有效的标识符，不区分大小写。所以在相互调用时，汇编源程序中的标识符不要超过 8 个字符（Turbo C 在计算机上没有这个限制），并按照 C 语言的习惯最好采用小写字母。

（2）声明约定

C 语言和汇编语言进行混合编程时，汇编子程序中公共的过程名及变量名应该用 public 进行说明：

```
public    _name
```

其中，name 是汇编子程序的一个过程名或变量名，需要注意的是在名字前要加下画线。而在 C 中应将在本程序中用到的汇编子程序的公共的过程名和变量名说明为外部符号，并且不能在名字前加下画线。

C 的说明形式为：

```
extern          返回值类型          函数名称（参数类型表）;
extern          变量类型       变量名;
```

例如 C 的说明形式：

```
extern          num;
extern          void addition(int *,int *);
```

经过说明后，C 语言程序可以使用汇编语言的子程序及变量。

（3）寄存器使用约定

在编写汇编子程序过程时，应注意寄存器使用方面的限制，不同的 C 语言版本中的规定有一些差别。一般来说（对 Turbo C），在汇编语言子程序过程中：AX、BX、CX、DX、ES 可以随意使用；BP、SP、SI、DI、CS、DS、SS 使用时先将原始内容保存入栈，在子程序结束前恢复其原始内容。

（4）存储模式约定

Turbo C 提供了 6 种存储模式，分别是：微型模式（Tiny）、小型模式（Small）、紧凑模式（Compact）、中型模式（Medium）、大型模式（Large）和巨型模式（Huge），与汇编程序相应的存储模式一样。

为了使汇编语言与 Turbo C 语言程序连接到一起，对于汇编语言简化段定义格式来说，两者必须具有相同的存储模式。汇编语言程序采用.model 伪指令，Turbo C 利用 TCC 选项“–m”指定各自的存储模式。相同的存储模式将自动产生相互兼容的调用和返回类型。同时，汇编程序的段定义伪指令.code、.data 等也将产生与 Turbo C 相兼容的段名称和属性。

2. 混合模块的编译和连接

根据以上约定，编写一 C 语言程序调用汇编语言子程序的简单例子。这里暂时不涉及参数传递的问题。

【例 11.3】用汇编语言编写冒泡排序程序（BubbleSort.asm），被排序的数组元素放在 C 语言编写的程序（BubbleSortMain.c）中输入，程序源代码如下：

```
/*排序的主函数 BubbleSortMain.c */
#include “stdio.h”
extern  void  sort(int *,int)
main()
{
        int    elem[5];
        int i,n;
        i=0;
        while(i<5)
        {
              scanf(%d,&elem[i]);
              i++;
        }
        sort(elem,5);
        printf(“排序后的结果：\n”);
        i=0;
        while(i<10)
        printf(“%5d”,elem[i++]);
}
;汇编语言程序 BubbleSort.asm
_TEXT   SEGMENT    WORD     ‘CODE’
          ASSUME     CS:_TEXT
          PUBLIC _SORT
_SORT       PROC      FAR                      ;C 程序为大模式，汇编用远过程
              PUSH      BP
              MOV       BP,SP
              PUSH      SI
              PUSH      DI                       ;保护现场
              MOV       DX,[BP+10]
              LES       DI,DWORD PRT[BP+6]       ;对于源程返回地址占用 4 字节，
                                                 ;BP 占用两字节
              MOVCX,DX                           ;所以第一个参数在 BP+6 的单元
              SHL       DX,1
              DEC CX
COTI：        PUSH      CX
              MOV       SI,0
              MOV       BX,DX
              DEC BX
              DEC BX
AGAIN：       MOV       AX,ES:[DI][BX]
```

```
                CMP     AX,ES:[DI][BX-2]        ;由于 C 为 INT 型占两个字节
                JGE     NEXT
                XCHG    AX,ES:[DI][BX-2]        ;交换
                MOVSI, –1
NEXT：          DEC BX
                DEC BX
                LOOP    AGAIN
                POP     CX
                CMP     SI,0
                JNE     COTI
                POP     DI
                POP     SI
                POP     SP                      ;恢复现场
                RET
_SORT           ENDP
_TEXT           ENDS
                END _SORT
```

编辑完成上述两个源程序文件之后，可以按如下步骤进行编译和连接：

（1）利用汇编程序编译汇编语言程序称为.obj 目标代码文件，例如：

ml/c　　BubbleSort.asm

它将生成 BubbleSort.obj 文件。Ml 的默认选项/cx 表示保持汇编语言程序中的名字的大小写不变，这样才能在连接时不出错；选项/cu 将使名字转变成大写，不应使用。

（2）利用 C 编译程序编译 C 语言程序称为.obj 目标代码文件，例如：

TCC-c　BubbleSortMain.c

其中，“–c”参数表示只是编译，不连接，结果生成 BubbleSortMain.obj 文件。TCC 默认采用小型存储模式。若采用其他模式，要利用“−m”选项。若 C 程序中有#include 包含文件行，则需加“−I”选项。

（3）利用连接程序将各个目标代码文件连接在一起，得到可执行程序文件，例如：

TLINK　lib\cos　BubbleSortMain　BubbleSort, BubbleSortMain.exe,,lib\cs

注意：直接使用 Turbo C 的连接程序 TLINK 进行连接时，用户必须指定要连接的与存储模式一致的初始化模块和函数库文件，并且初始化模块必须是第一个文件。

编译和连接也可以利用命令行一次完成，这样更加方便。它的一般格式为：

TCC –mx –I 包含文件路径−L 库文件路径　filename1　filename2 …

【例 11.3】可以利用如下命令：

TCC –ms –Iinclude –Llib　BubbleSortMain.c BubbleSort.obj

C 程序的编译和模块连接也可以在 Turbo C 集成环境下进行。此时需要选用一致的存储模式，并建立一个工程文件。包括需变异连接的 C 源文件以及汇编语言目标文件名。

3. 混合编程的参数传递

参数传递常常是编制子程序时一个最为重要的问题，在传递参数过程中要求参数个数、

类型、顺序一一对应，编程时应该正确应用参数传递的方法。下面说明 C 语言和汇编语言程序参数传递的方法和注意事项。

1）利用堆栈传递参数

C 语言程序可以通过堆栈将参数传递给被调用函数。

（1）C 程序调用函数之前，先从该函数中的最右边的参数开始依次将参数压入堆栈，最后压入最左边的参数。也就是说，参数压入堆栈的顺序与实参表中参数的排列顺序相反，即从右到左，第一个参数最后压入堆栈。当被调用函数运行结束后，压入堆栈中的参数已无保留的价值，C 程序会立即调整堆栈指针 SP，使之恢复压入参数以前的值，这样就释放了堆栈中为参数保留的空间。也就是说，堆栈的平衡是在主函数程序实现的，子程序不必在返回时调整堆栈指针 SP。这就是参数传递的 C 语言规则。

（2）C 语言参数在压栈时，有些参数先要进行数据类型转换再压栈，各类型的参数在堆栈中所占的字节数不相同，如 int 占 2 个字节、real 占 8 字节等，表 11.1 中给出压栈参数的类型与所占的字节数。

表 11.1　参数类型占用字节数

参数的数据类型	占用字节数
字符型、整型、近指针、指向数组的指针	2 字节
长整型、远指针（先段地址、后偏移地址）	2 字节
双精度型、浮点型（先转换为双精度型）	8 字节
结构体	N 字节

2）返回值的传递

被调用函数的返回值，按下列规则传递给调用者：

（1）如果返回值小于或等于 16 位，则将其存放在 AX 寄存器中。

（2）如果返回值等于 32 位，则存放在 DX.AX 寄存器中，其中 DX 存储高 16 位，AX 存储低 16 位。

（3）如果返回值大于 32 位，则存放在静态变量存储区；AX 寄存器存放指向这个存储区的偏移地址；对于 32 位 far 指针，则还利用 DX 存放段地址。

3）地址参数的传递

C 语言程序的参数传递，可采用传值和传址两种方式。如果参数是传值的，可直接写出实际参数。如果参数是地址的，则在说明中，将参数类型说明为指针型。调用时，用“&”取得变量的地址作为实参传递。在汇编语言子程序中，利用基址指针 BP，先取得地址，再间接取内容，修改后送回原处，同时以 RET 返回。

11.2　汇编语言在Visual C++中的应用

C++语言是 C 语言的超集，它是在 C 语言的基础上扩展形成的面向对象程序设计语言。微软的 Visiual C++则是 Windows 平台上广泛应用的开发系统。本节以 Visual C++6.0 为例，说明 32 位 Windows 环境下汇编语言与 C++的混合编程。它也可以分为嵌入式和模块调用两种。

11.2.1 嵌入汇编语言指令

Visual C++直接支持嵌入式汇编方式，不需要独立的汇编系统和另外的连接步骤。所以，嵌入式汇编比模块连接方式更简单方便。Visual C++的嵌入式汇编方式与其他 C/C++编译系统的基本原理是一样的，当然有些细节上的差别。嵌入汇编指令采用__asm 关键字（注意，asm 前是两个下画线。但 Visual C++5.0/6.0 也支持一个下画线的格式_asm，目的是与以前版本保持兼容）。

Visual C++嵌入式汇编格式__asm{指令}是采用花括号的汇编语言程序段形式，例如：

```
/*__asm 程序段*/
__asm
{
        mov     eax,01h             //支持 C++语言的注释格式
        mov     dx,0xD007           ;0xD007=D007h,支持 C/C++的表达式
        out     dx,eax
}
```

也具有单条汇编语言指令形式：

```
//单条__asm 汇编指令格式
__asm   mov     eax,01h
__asm   mov     dx,0D007h
__asm   out     dx,eax
```

另外，还可以使用空格在一行分隔多个__asm 汇编语言指令

```
__asm   mov     eax,01h __asm   mov     dx,0D007h __asm   out     dx,eax
```

上面 3 种形式产生相同的代码，但第一种形式具有更多的优点。因为它可以将 C++代码与汇编代码明确分开，避免混淆。如果将__asm 指令和 C++语句放在同一行且不使用括号，编译器就分不清汇编代码到什么地方结束和 C++语句从哪里开始。__asm 花括号中的程序段不影响变量的作用范围。__asm 块允许嵌套，嵌套也不影响变量的作用范围。

1. 在__asm中使用汇编语言的注意事项

（1）嵌入式汇编代码支持 80486 的全部指令系统。Visual C++还支持 MMX 指令集。对于还不能支持的指令，Visual C++提供了_emit 伪指令进行扩展。

（2）Visual C++不支持 MASM 的宏伪指令（如 MARO、ENDM、REPEAT/FOR/FORC 等）和宏操作符（如!、&、%等）。

（3）嵌入式汇编代码可以使用 MASM 的表达式，这个表达式是操作数和操作符的组合，产生一个数值或地址。

（4）嵌入式汇编行还可以使用 MASM 的注释风格。

（5）嵌入式汇编代码可以使用 LENGTH、SIZE、TYPE 操作符来获取 C++变量和类型的大小。LENGTH 用来返回数组元素的个数，对非数组变量返回值为 1。TYPE 返回 C++类型或变量的大小，如果变量是一个数组，它返回数组单个元素的大小。SIZE 返回 C++变量的大小，即 LENGTH 和 TYPE 的乘积。

例如，对于数据 int iarray[8] （int 类型是 32 位，4 个字节），则：

LENGTH　　iarray 返回 8

TYPE　　iarray 返回 4

SIZE　　iarray 返回 32

（6）在用汇编语言编写的函数中，不必保存 EAX、EBX、ECX、EDX、ESI 和 EDI 寄存器，但必须保存函数中使用的其他寄存器（如 DS、SS、ESP、EBP 和整数标志寄存器）。

（7）嵌入式汇编引用段寄存器时应该通过寄存器而不是通过段名，段超越时，必须清晰地用段寄存器说明，例如 ES：[EBX]。

2. 在__asm使用C++语言的注意事项

嵌入式汇编代码可以使用 C++的下列元素：符号（包括标号、变量、函数名）、常量（包括符号常量、枚举成员）、宏和预处理指令，注释（/*……*/和//，也可以使用汇编语言的注释风格）、类型名及结构、联合的成员。

嵌入式汇编语句使用 C++符号也有一些限制：每一个汇编语言语句只包含一个 C++符号（包含多个符号只能通过使用 LENGTH、TYPE 和 SIZE 表达式）。__asm 中引用函数前必须在程序说明其原型(否则编译程序将分不出是函数名还是标号)。__asm 中不能使用和 MASM 保留字（）相同的 C++符号，也不能识别结构 Structure 和联用 Union 关键字。

嵌入式汇编语言语句中，可以使用汇编语言格式表示整数常量（如 378h），也可以采用 C++的格式（如 0x378）。

嵌入式汇编语言语句不能使用 C++的专用操作符，如<<。对两种语言都有的操作符在汇编语句中作为汇编语言操作符，如*、[ ]。例如：

```
int        array[6];        //C++语句中，[]表示数组的某个元素
__asm    mov        array[6],bx     //汇编语言中、[]表示距离标识符的字节偏移量
```

嵌入式汇编中可以引用包含该__asm 作用范围内的任何符号（包括变量名），它通过使用变量名引用 C++的变量。例如：若 var 是 C++中的整型（int）变量，则可以使用如下语句：

```
__asm    mov        eax，var
```

嵌入式汇编中的标号和 C++的标号相似。它的作用范围为定义它的函数中有效。汇编转移指令和 C++的 goto 指令都可以逃到__asm 块内或块外的标号。

__asm 块中定义的标号对大小写不敏感，汇编语言指令跳到 C++中的标号也不分大小写，C++中的标号只有使用 goto 语句才对大小写敏感。

利用 C/C++宏可以方便地将汇编语言代码插入源程序中。C/C++宏将扩展成为一个逻辑行，所以书写具有嵌入汇编的 C/C++宏时，应遵循下列规则：将__asm 程序段放在括号中，每一个汇编语言指令前必须有__asm 标志，应该使用 C 的注释风格（/*　　*/），不要使用 C++的单行注释和汇编语言的分号注释（;）方式。例如：

```
#define      portio      __asm                    \
/*Port   output*/                                 \
{                                                  \
        __asm     mov       eax,01h          \
        __asm     mov       dx,0xD007        \
        __asm     out       dx,eax           \
}
```

该宏展开为一个逻辑行（其中“\”是续行符）：

_ _asm /*Portoutput*/{_ _asm mov eax,01h _ _asm mov dx,0xD007 _ _asm out dx,eax }

3. 用_ _asm程序段编写函数

采用嵌入式汇编书写函数，较模块调用更加方便，因为这不需要利用独立的汇编程序，而且给函数传递参数和从函数返回值也非常方便。

嵌入式汇编不仅可以编写 C/C++函数，还可以调用 C 函数（包括 C 库函数）和非重载的全部 C++函数，也可以调用任何用 extern “C” 说明的函数，但不能调用 C++的成员函数。因为所有的标准头文件都采用 extern “C”说明库函数，所以 C++程序中的嵌入式汇编可以调用 C 库函数。

【例 11.4】用嵌入式汇编编写函数。

```
#include<iostream.h>
int power2(int,int);
void        main(void)
{
        cout<<”2 的 6 次方乘 5 等于：\t”;
        cout<<power2(5,6)<<endl;
}
int power2(int num,int power)
{
        _ _asm
        {
        mov   eax,num                   ;取第一个参数
        mov   ecx,power                 ;取第二个参数
        shl   eax,cl                    ;计算 EAX:EAX *(2^cl)
        }                               //返回值存于 EAX
}
```

11.2.2 调用汇编语言过程

采用模块调用方式，在 C++语言中调用汇编语言过程和 C 语言调用汇编过程类似，同样要遵守名称、调用、参数传递与返回等约定。

1. 采用一致的调用规范

C/C++与汇编语言混合编程的参数传递通常利用堆栈，调用规范决定利用堆栈的方法和命名约定，两者要一致，例如 Visual C++的_cdecl 调用规范与 MASM 的 C 语言类型。

Visual C++语言具有 3 种调用规范：_cdecl、_stdcall 和_fastcall。Visual C++默认采用_cdecl 调用规范，它在名字前自动加一个下画线，从右到左将实参压入堆栈，由调用程序进行堆栈的平衡。

Windows 图形用户界面过程和 API 函数等采用_stdcall 调用规范，它在名字前自动加一个下画线，名字后跟@和表示参数所占字节数的十进制数值，从右到左将实参压入堆栈，由被调用程序平衡堆栈。

Visual C++的_fastcall 调用规范是在名字前、后都加一个@，后再跟表示参数所占字节数的十进制数值。首先利用寄存器 ECX、EDX 传递前两个字节参数，其他参数再通过堆栈传递（从右到左），由被调用程序平衡堆栈。与其他语言进行混合编程时不要使用_fastcall 规范。

2. 申明公用函数和变量

对于 C++语言和汇编语言的公用过程名、变量名应该进行申明，并且标识符一样。注意 C++语言对标识符区分字母的大小写，而汇编语言不区分大小写。

在 C++语言程序中，采用 extern“C”{}对所要调用的外部过程、函数、变量予以说明，说明形式如下：

extern“C”{返回值类型 调用规范 函数名称（参数类型表）；}

extern“C”{变量类型 变量名；}

汇编语言程序中供外部使用的标识符应具有 PUBLIC 属性，使用外部标识符要利用 EXTERN 申明。

3. 入口参数和返回参数的约定

C/C++语言中不论采用何种调用规范，传递参数的形式都是“传值”的（by value），但除了数组（因为数组名表示的是第一个元素的地址）。参数“地址”（by reference）应利用指针数据类型。

Visual C++与 MASM 数据类型对应关系如表 11.2 所示。但不论何种整数类型，进行参数传递时，都扩展成 32 位。需要注意，32 位 Visual C++版本中整型（int）类型是 4 字节。另外 32 位 Visual C++中没有近、远调用之分，所有调用都是 32 位的偏移地址，所有的地址参数也都是 32 位偏移地址，在堆栈中占 4 字节。

表 11.2 Visual C++与 MASM 数据类型对应关系

Visual C++的数据类型	MASM 的数据类型	Visual C++的数据类型	MASM 的数据类型	字节数
unisigned char	BYTE	Char	SBYTE	1
Unisigned short	WORD	Short	SWORD	2
Unisigned long[int]	DWORD	Long[int]	SDWORD	4
Float	REAL4			4
Double	REAL8			8
Long double	REAL10			10

参数返回时 8 位值在 AL 返回，16 位值在 AX 返回，32 位值存在 EAX 寄存器返回，64 位返回值存放在 EDX.EAX 寄存器中，更大数据则将它们的地址指针存放在 EAX 中返回。

4. 编写汇编语言程序要注意的问题

在编写与 Visual C++6.0 混合编程的汇编语言过程时，程序员必须明确这是一个 32 位的编程环境，程序员可以采用全部 32 位 Intel80x86CPU 指令，但必须首先留心 32 位指令程序设计的问题，例如用.386p 等处理器伪指令说明采用的指令集，有些指令在 32 位环境与在 16 位 MS-DOS 环境存在差别等。

对于 Visual C++的 32 位程序来说，没有存储模式的选择；汇编语言简化段定义格式应该采用平展模式（flat），并且汇编时采用选项/coff。ML 命令行的选项/coff 使得产生的.obj 模块

文件采用与 32 位 Microsoft Windows NT 兼容的 COFF 格式。不要修改 ML 的默认选项/Cx（表示保持汇编语言程序中的名字的大小写不变）。

另外，32 位编程环境的寄存器是 32 位的，所以汇编语言过程存取堆栈要使用 32 位寄存器 EBP 进行相对寻址，如：mov eax，[ebp+3]，而不能采用 BP。

11.2.3 使用汇编语言优化C++代码

汇编语言的优势之一是生成的代码运行速度快，所以其用途之一就是优化高级语言中运行次数多、速度要求高的关键程序段。下面举例说明利用汇编语言程序优化 C++代码。

【例 11.5】现有一程序，要求在一个较大的数组 array[10000]中查找某个指定的数值，假设全部元素都为 0，要查找的数值为 100。优化前的代码如下：

```
#include <iostream.h>
bool findArray(int searchVal int array[] ,int count);
void main (void)
{
        const int SIZE=10000;
        int array[SIZE];
        int temp1,temp2;
        for(int i=0;i<=SIZE;i++) array[i]=0;      //将数组的所有元素赋值为 0
        _ _asm{                                   //保存当前的时钟周期数
              rdtsc
              mov     temp1,eax
              mov     temp2,edx
        }
        findArray(100,array,SIZE);
        _ _asm{                                   //计算程序使用周期数
              rdtsc
              sub    eax,temp1
              sub    edx,temp2
              mov temp1,eax
              mov temp2,edx
        }
        Cout<<"程序执行的时钟周期数："<<temp1+temp2*(2^32)<<endl;
}
bool findArray(int searchVal,int array[],int count)
{
        for(int i=0 ;i<count;i++)
        if(searchVal==array[i])
              return true;
     return false;
}
```

将该程序在 Visual C++集成开发环境下采用调试（DEBUG）版本进行编译、连接生成可执行文件。然后再命令行模式下运行生成的可执行文件，就可以显示执行 findArray 过程需要的时钟周期数。程序运行的速度当然与及其有关，另外，因为现代 PC 存在高速缓冲存储器 Cache，程序的执行是一个动态过程，所以该程序在不同的 PC 上运行显示的时钟周期数是不相同的，即使在同一台机器上多次运行该时钟周期数也不相同。例如在某台采用 Pentium4 微处理器、时钟频率为 1.8GHz 的 PC 上显示的是约为 81000。

DEBUG 版本的可执行程序文件是没有经过优化的，Visual C++使用的编译程序 CL.EXE 支持许多优化参数，例如以 O 开头的参数都是优化参数。在项目配置采用调试（DEBUG）版本时，默认不进行优化，对应参数“/Od”。在项目配置采用发布（Release）版本时，对应参数“/O2”，它按照最快运行速度的原则进行优化（Maximize Speed）。还有参数“O1”是按照最小空间的原则优化（Maxisize Size）。它们都可以通过 Visual C++集成开发环境的工程（Project）菜单的设置（Setting）命令进行设置。

现在执行创建菜单的设置活动配置（Set Active Configuration）命令选择 Release 版本，重新进行编译和连接，生成经过编译器优化的 Release 版本的可执行文件。在同一台 PC 上运行，显示的时钟周期数约为 31000。由此可见，程序运行速度提高了 2.5 倍以上，编译器优化的效果是很可观的。现在再用嵌入式汇编语言编写 findArray 过程，代码如下：

```
bool findArray(int search)
{
    _ _asm{
        mov ecx,count
        jecxz notfound                    ;如果数组元素个数为 0，则退出
        mov edi,array
        mov eax,searchVal
    again: cmp eax,[edi]
        je found
        add edi ,4
        loop   again
    notfound: xor al,al
        jmp done
    found: mov al,1
    done:
    }
}
```

用这个过程代替 C++代码再次生成可执行文件。该可执行文件在同一台机器上运行的结果与前面发布的 Release 版本不相上下。可见，简单的汇编语言程序与没有优化的程序相比其速度也可以取得可观的提高。

11.2.4 使用Visual C++开发汇编语言程序

功能强大的集成开发环境 Visual C++能够用来编辑、汇编、连接和调试汇编语言程序。下面，简单描述如果利用 Visual C++集成开发环境开发和调试汇编语言程序，并说明其中应

该注意的问题。

1. 使用Visual C++开发汇编语言程序的开发过程

（1）创建项目

新建一个工程项目，根据需要选择 32 位控制台应用程序（Win32 Console Application）或 32 位窗口应用程序（Win32 Application）。输入工程项目所在的磁盘目录，输入工程名称，并选择一个空白工程（An Empty Project）。

（2）创建汇编语言源程序文件

选择文本文件（Text file），输入源程序文件名以及扩展名 asm。然后将该文件添加到工程项目中来。

（3）配置环境

在文件视图（File View）选中上一步创建的汇编语言源程序文件，选择右击弹出的设置（Setting）命令，或者通过工程菜单的设置命令展开其工程设置窗口。在右边选择定制创建（Custom Build）标签，在其命令（Commands）文本框输入进行汇编的命令；还要在其输出（Outputs）文本框输入汇编后目标模块文件名。文件名也可以单击该标签下面的文件（Files）选项选择 Input Name，另外，应该事先将 ML.EXE 和 ML.ERR 文件复制到 Visual C++所在的 bin 目录下；或者在输入汇编命令的同时输入 ML.EXE 所在的目录路径。

这时，可以调用创建菜单的创建命令进行汇编语言程序的汇编和连接。汇编连接的有关信息显示在下面输出（Output）窗口的创建视图中。如果程序正确无误，会生成可执行文件（默认是调试版本，在 DEBUG 目录下）。如果源程序有错误，创建视图将显示错误所在的行号以及错误的原因。双击错误信息，光标将定位到出现错误的源程序行。

2. 使用Visual C++开发汇编语言程序的调试过程

为了使 Visual C++集成开发环境更适合对汇编语言程序的调试，可以通过工具（Tools）菜单的选项（Options）命令展开调试（Debug）标签页进行设置。通用（General）下的十六进制显示（Hex 阿的 cimal display）应该选中，以便十六进制形式显示输入/输出数据（此时，可以用 On 开头表示输入十进制数据）。反汇编窗口（Disassembly window）下要选中代码字节（Code bytes）。存储器窗口（Memory window）下选中固定宽度（Fixed width），并在后面填入数字 16。

在文件视图（FileView）中双击源程序文件名，则编辑窗口将显示这个源程序。移动光标到需要暂停的语句行，按 F9 键，就在该行设置了一个断点（前面有一个红色的圆点）；光标在已经设置断点的语句行时再次按 F9 键，则取消断点。一个程序可以设置多个断点。

使用创建菜单的执行命令（快捷键 Ctrl+F5）可以运行已经编译连接的可执行程序，也可以从创建菜单的开始调试（Start Debug）命令选择在调试状态下执行程序，例如运行（Go，其快捷键是 F5）。常用的调试命令还有不跟踪子程序的单步执行（Step over，快捷键是 F10），跟踪子程序的单步执行（Step Into，快捷键是 F11）等。进入调试状态后，原来的创建菜单改变成为调试菜单。

如果程序设置了断点，启动程序运行后将停留在断点语句行，在源程序窗口中有一个黄色箭头指示。这时，利用视图（View）菜单的调试窗口（Debug windows）命令，就可以打开各种窗口观察程序当前的运行状态。

11.3 汇编语言与C/C++的混合编程实例

【例 11.6】C 语言和汇编语言混合编程实例：用嵌入式汇编语言编写一个阶乘函数。

分析：在本例中利用嵌入式汇编编写计算阶乘的函数。

```
#include<iostream.h>
int fac(int);
void main(void)
{
        int n;
        cin>>n;
        cout<<n<< "!=";
        cout<<fac(n)<<endl;              //调用嵌入式汇编阶乘函数 fac 计算 n!
}
int fac(int num)
{
__asm{
        mov     eax,1
        mov     ecx,num                  //获取入口参数
        mov     ebx,1
again: mul   ebx                         //计算 n!
        inc   ebx
        loop  again                      //循环结束后，由 eax 返回阶乘结果
}
}
```

【例 11.7】已知在 CMOS 中，偏移地址 04H 存放着系统时间的小时数，02H 处存放系统时间的分钟数，00H 存放着系统时间的秒数，存放的形式均为压缩型 BCD 码。利用汇编程序访问 CMOS 读取系统时间，C 程序显示系统时间。

显示系统时间 CMOS.C 文件如下：

```
#include "bios.h"
extern   int   *f1();
main()
{
        int     h,m,s;
        int     *a;
        while(1)
  {
        a=f1();
        h=*a;                            /*从返回值中读出系统时间*/
```

```
        m=*(a+1);
        s=*(a+2);
        h=h/16+h%16;                    /*将压缩型 BCD 转换成十进制*/
        m=m/16+m%16;
        s=s/16+s%16;
        printf("%02d:%02d:02d\r",h,m,s);
        if(kbhit()) break;              /*如果按键，则结束程序*/
    }
}
```

读取 CMOS 中系统时间的汇编程序 CMOS.ASM 文件如下：

```
.MODEL SMALL
.DATA
TIME   DW   ?,?,?                      ;存放系统时间
.CODE
PUBLIC     _F1
_F1        PROC
           CLI
           MOV      AL,4
           OUT      70H,AL
           IN       AL,71H             ;读取小时数
           MOV      AH,0
           MOV      TIME,AX
           MOV      AL,2
           OUT      70H,AL
           IN       AL,71H             ;读取分钟数
           MOV      TIME+2,AX
           MOV      AL,0
           MOV      70H,AL
           IN       AL,71H             ;读取秒数
           MOV      TIME+4,AX
           LEA      AX,TIME            ;返回地址值
           STI
           RET
       _F1 ENDP
           END
```

【例 11.8】Visual C++与汇编语言混合编程实例：本例将演示 Visual C++调用嵌入式汇编函数和外部汇编语言过程。

分析：本例中嵌入式汇编语言函数 imin()实现查找数组 iarray 中的最小数，而通过外部汇编过程 isum 实现对数组 iarray 的求和。

```
//C++源程序
#include<iostream.h>
extern “C”{long    isum(int,int *);}
int imin(int,int *);
void main(void)
{
        const int SIZE=10;
        int array[SIZE];
        int temp;
        cout<<”请输入 10 个整数(-214748364~214748364 之间)：”<<endl;
        for(temp=0;temp<SIZE；temp++)
              cin>>array[temp];                    //输入 10 个数据
        cout<<endl;
        cout<<”整数数据之和：\t”<<isum(SIZE,array)<<endl;
        cout<<”其中最小数为：\t”<<imin(SIZE,array)<<endl;

}
int imin(int itmp,int iarray[])                    //求 itmp 个元素的数组 iarray 的最小数
{
        __asm{
                    mov ecx,itmp
                    jecxz minexit                  ;如果个数为 0，则返回
                    dec ecx
                    mov esi,iarray
                    mov eax,[esi]
                    jecxz minexit                  ;个数为 1，则返回
        minlp:      add esi,4
                    cmp eax,[esi]                  ;比较两个数据的大小
                    jle nochange
                    mov eax,[esi]                  ;取得较小值
        nochange:   loop minlp
        minexit:
    }
}
;汇编语言程序
                    ;NAME eaxmple11_1.asm
                    .386P
                    .MODEL FLAT,C
                    .CODE
;32 位有符号数据的求和过程
```

```
ISUM    PROC USES ECX ESI,\
            COUNT:DWORD,DARRA:PTR   ;“\”为续行符
            MOV ECX,COUNT           ;个数为 0，和为 0
            XOR EAX,EAX
            JECXZ SUMEXIT
            MOV ESI,DARRAY          ;个数为 1，和为本身
            MOV EAX,[ESI]
            DEC ECX
            JECXZ SUMEXIT
SUMLP:      ADD ESI,4
            ADD EAX,[ESI]
            LOOP SUMLP
SUMEXIT:    RET
ISUM    ENDP
            END
```

习　题

11.1　什么是混合编程？汇编语言与 C/C++语言的混合编程有哪两种方法？各有什么特点？

11.2　Turbo C 能够嵌入的汇编语言指令有哪些？Visual C++支持的汇编指令集又是什么？

11.3　说明 Visual C++嵌入汇编语言指令的形式。

11.4　如下有一个简单的 C 语言源程序：

```
main()
{
    sum(3,4);
}
sum(int x,int y)            /*求两数之和的函数，x 和 y 是入口参数*/
{
    int sum;                /*定义一个局部变量*/
    sum=x+y;                /*求和*/
    return(sum);            /*返回和值*/
}
```

要求用嵌入汇编方法实现加法函数 sum，上机调试通过。

11.5　利用 11.4 提供的 C 程序，编写一个汇编语言子程序实现加法函数 sum，并用模块连接方式实现与 C 语言主函数混合编程，上机调试通过。

11.6　用汇编语言编写函数 Display(Data)，其功能是在当前光标处显示无符号整数 Data 然后编写一个 C 语言程序调用 Display 来显示整型变量的值。

11.7　说明如下程序的输出结果，然后上机验证。

```
//C++程序
#include <iostream.h>
Extern “C”{
void MLSub(char *,short *,long *);
}
char chararray[4]=”abc”;
short shortarray[3]={1,2,3};
long longarray[3]={32768,32769,32770};
void main(void)
{
        cout<<chararray<<endl;
        cout<<shortarray[0]<< shortarray[1]<< shortarray[2]<<endl;
        cout<<longarray[0]<< longarray[1]<< longarray[2]<<endl;
        mlsub(chararray,shortarray,longarray);
        cout<<chararray<<endl;
        cout<<shortarray[0]<< shortarray[1]<< shortarray[2]<<endl;
        cout<<longarray[0]<< longarray[1]<< longarray[2]<<endl;
}
;汇编语言程序
.386
.MODLE flag,C
.CODE
MLSub PROC USES ESI,\
                ARRAYCHAR:PTR,ARRAYSHORT:PTR,ARRAYLONG:PTR
                MOV ESI,ARRAYCHAR
                MOV BYTE PTR[ESI],”x”
                MOV BYTE PTR[ESI+1],”y”
                MOV BYTE PTR[ESI+2],”z”
                MOV ESI,ARRAYSHORT;
                ADD WORD PTR[ESI],7
                ADD WORD PTR[ESI+2],7
                ADD WORD PTR[ESI+4],7
                MOV ESI,ARRAYLONG;
                INC DWORD PTR[ESI]
                INC DWORD PTR[ESI+4]
                INC DWORD PTR[ESI+8]
                RET
MLSUB       ENDP
                END
```

附录A　上 机 实 验

与高级语言相比，对汇编语言程序设计方法和技巧的掌握更多地依赖于对处理器及存储器的了解，程序的调试需要更多地对当前相关寄存器和存储单元中数据的了解。因此，上机操作是学习汇编语言程序设计方法和技巧的重要环节。

本实验指导旨在通过精选的实验内容使学习者能够验证所学的程序设计方法和技巧，进而使用这些方法和技巧进行程序设计的实践。为此，给出的每个实验均有验证和设计这两类实验题。

实验一　程序的编辑、汇编、连接和调试

一、目的和要求

通过对一个简单程序的编辑、汇编、连接及调试，学习汇编语言程序设计上机操作的基本方法，为以后各项实验，以及为实际的程序设计建立基础。

二、实验内容

1. 验证题

对【例 5.7】给出的程序进行编辑、汇编、连接和调试。要求通过 DEBUG 工具检查每一条算术运算指令执行后相关寄存器的内容，检查程序执行前后各变量的内容。

2. 设计题

（1）编写一程序，做 BCD 数的四则运算，在程序中设置非压缩 BCD 数 *X*，从键盘输入一位十进制数 *Y*。将 *X*+*Y*、*X*−*Y*、*X***Y* 及 *X*/*Y*（不考虑余数）分别送 ANS_A，ANS_S，ANS_M 及 ANS_D 变量。上机调试程序，检查执行结果。

（2）编写一程序，实现两个三字节无符号数 DATA1 和 DATA2 的相加，结果送四字节变量 ANS。上机调试程序，检查执行结果。

实验二　分支程序设计

一、目的和要求

（1）进一步了解转移指令的格式和功能，通过解决分支问题练习条件转移指令的选用及无条件转移指令的使用。

（2）练习针对具体问题建立合适的分支程序结构，并掌握分支程序的调试方法。

二、实验内容

1. 验证题

调试【例 6.14】给出的程序。要求针对两数据块相对位置的三种情况分别验证程序正确性。

2. 设计题

（1）在 BUF 数据区中放有三个双字节数，现要求将这三个数按从大到小次序重新存放。分别将这些数视为有符号数和无符号数这两种情况编写程序。上机调试程序，检查执行结果。

（2）分别采用测试法和跳转表法编程解决第 6 章习题 8 提出的问题。上机调试程序，并从程序调试角度来说明，解决这类问题时宜选用哪一种分支程序设计方法。

实验三　循环程序设计

一、目的和要求

（1）进一步了解重复控制指令、串操作指令及重复前缀的格式和功能，通过解决循环问题练习这些指令和重复前缀的选用。对于既可以用重复控制指令，又可以用串操作指令及重复前缀的场合，比较两者的优劣。

（2）练习针对具体问题建立合适的循环结构，进一步了解计数控制、条件控制的循环结构的适用场合。

二、实验内容

1. 验证题

调试【例 7.10】和【例 7.11】给出的程序，验证两个程序的正确性。对于这两个程序在 SI、DI 初值设置上的区别做出分析，并对两个程序的优劣做比较。

2. 设计题

（1）编写程序，以统计 BUF 数据区中各有符号字节数的平均值。上机调试程序，检查程序正确性。

（2）按照第 7 章习题 6 要求编程，并且要求在搜索第一个零数据时分别采用重复控制指令和串操作指令。上机调试程序，检查程序正确性，并指出该程序使用了什么循环控制方法。

实验四　子　程　序

一、目的和要求

（1）进一步熟悉子程序调用、返回指令的使用方法，以及子程序的定义格式。

（2）加深对参数传递方法、现场保护和恢复的理解。

（3）练习针对具体问题采用不同方法解决子程序调用和返回中的参数传递问题，针对具体问题的需要对现场进行保护和恢复。

二、实验内容

1. 验证题

调试【例 8.7】、【例 8.8】及【例 8.9】给出的程序。要求对三个程序均在子程序开始位置检查入口参数，在子程序返回后检查出口参数。

2. 设计题

（1）按照第 8 章习题 7 要求编程。另外，要求分别采用三种方法实现参数传递，并要求保护和恢复现场。上机调试程序，检查程序正确性。

（2）设 BUF 数据区中有 n 个无符号字节数，试计算这些数之和，并以以下形式显示：

$$d_0+d_1+\cdots+d_{n-1}=S$$

其中 d_i 为第 i 个数的十进制表示，S 为和值的十进制表示。编写程序，要求将十进制数的显示使用子程序 SUB1 实现，求和工作用子程序 SUB2 实现。上机调试程序，检查程序正确性。

实验五　高级汇编语言技术

一、目的和要求

（1）进一步掌握宏汇编、重复汇编及条件汇编的一般格式，进一步熟悉相关伪指令的功能和使用方法。

（2）加强对宏汇编与子程序异同点的理解。

（3）练习针对具体问题灵活使用高级汇编语言技术编写简洁高效的源程序。

二、实验内容

1. 验证题

通过对【例 9.2】给出的宏定义的宏调用，实现寄存器 AX、BX 及 CX 内容乘以 10 的功能；验证【例 9.13】、【例 9.14】中重复汇编结构的功能；验证【例 9.20】给出的宏定义的功能。

2. 设计题

（1）使用重复汇编的方法建立一个数据表，其中存放 1，2，3，6，…，Σi，根据给定的 i 值，查表求取表中第 i 项数据。编写程序，上机调试，检查程序正确性。

（2）按照第 9 章习题 8 的要求编程序。另外，要求将宏指令的功能用子程序实现。对上述两个程序进行调试，验证其正确性，并且对两个程序的优缺点做出分析。

实验六　DOS功能调用与BIOS中断调用

一、目的和要求

（1）进一步熟悉汇编语言的编程，加深对中断调用过程的理解，并进一步掌握 BIOS 中断调用、DOS 系统功能调用的方法，并在此基础之上具备编写简单程序的能力。

（2）在了解 DOS 功能调用方法的基础之上，掌握与文件管理相关的知识。

（3）在了解 BIOS 功能调用方法的基础之上掌握通过 BIOS 中断调用读取磁盘扇区的方法。

二、实验内容

1. 验证题

调试【例 10.1】、【例 10.2】和【例 10.3】给出的程序，并验证此程序的正确性。

2. 设计题

（1）按照习题 10.3 要求编程。上机验证所编写程序的正确性。

（2）按照习题 10.4 和习题 10.5 要求编程。上机验证所编写程序的正确性。

实验七　C/C++语言与汇编语言的混合编程

一、目的和要求

通过对汇编程序和 C/C++的编辑、编译、连接及调试，初步掌握汇编语言程序与 C/C++程序设计连接的基本方法，掌握生成可执行文件的过程。

二、实验内容

1. 验证题

（1）对【例 11.1】和【例 11.2】给出的程序进行编译、连接并上机进行验证。

（2）分析习题 11.7 程序的功能，并上机进行验证。

2. 设计题

（1）按照本书习题 11.5 要求编程。上机验证所编写程序的正确性。

（2）按照本书习题 11.6 要求编程。上机验证所编写程序的正确性。

（3）改写【例 11.2】中的程序，要求程序完成从大写转换成小写功能并上机验证所编程序的正确性。

附录B　ASCII码表

编　码	字　符	编　码	字　符	编　码	字　符	编　码	字　符
00	NUL	20	SPACE	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	“	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	‘	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

注：①“编码”是十六进制数；　② SPACE 是空格；

③ LF=换行，FF=换页，CR=回车，DEL=删除，BEL=振铃。

附录C　80x86 指令表

说明：1. 凡助记符后标有数字 x 者，均表示所指定的 80x86 指令或以上处理器指令，否则为所有 80x86 及 Pentium 指令。

2. 标志位符号含义如下：0—置 0；1—置 1；x—根据结果设置；-—不影响；

U—无定义；R—恢复原先保存的值。

一、数据传送类				
指 令 名 称		操作数形式	功 能 说 明	标志位 ODITSZAPC
传送指令 MOV		寄，寄 寄，存 存，寄 段寄，寄 段寄，存 寄，段寄 存，段寄 寄，数 存，数	源操作数送目的操作数	_ _ _ _ _ _ _ _
交换字节顺序指令 BSWAP（4）		寄 32	设寄存器中字节编号如下：3-2-1-0 0 与 3 字节交换，1 与 2 字节交换	_ _ _ _ _ _ _ _
入栈指令	PUSH	寄 段寄存器 存	源操作数送堆栈指针所指栈顶单元	
	PUSH（2）	数	立即数送堆栈指针所指栈顶单元	
压入通用寄存器指令 PUSHA（2）			把通用寄存器 AX、CX、DX、BX、BP、SI 及 DI 顺序压入栈中	
压入 32 位通用寄存器指令 PUSHAD（3）			把通用寄存器 EAX、ECX、EDX、EBX、EBP、ESI 及 EDI 顺序压入栈顶单元	
出栈指令 POP		寄 段寄（除 CS 外） 存	栈顶内容弹出到目的操作数中	_ _ _ _ _ _ _ _
弹出通用寄存器指令 POPA（2）			从当前栈顶开始，把栈内容按顺序弹出到通用寄存器 DI、SI、BP、BX、DX、CX 及 AX 中	
弹出 32 位通用寄存器指令 POPAD（3）			从当前栈顶开始，把栈内容按顺序弹出到 32 位通用寄存器 EDI、ESI、EBP、EBX、EDX、ECX 及 EAX 中	
交换加指令 XADD（4）		寄，寄 存，寄	目的操作数=目的操作数+源操作数，源操作数地址=原来目的操作数地址	
交换指令 XCHG（4）		寄，寄 存，寄 寄，存	源操作数与目的操作数互换	_ _ _ _ _ _ _ _

续表

指令名称		操作数形式	功能说明	标志位 ODITSZAPC
输入指令 IN		累，数 累，DX	端口中信息送累加器，“数”为端口号，其值在 0~255 间。超过 255 的端口号放在 DX 中，累加器可为 AL 或 AX 或 EAX（80386 及其以后 CPU 可用）	_ _ _ _ _ _ _ _ _
输出指令 OUT		数，AX DX，AX	累加器的内容送端口，其余规定与 IN 指令同	_ _ _ _ _ _ _ _ _
字符转换指令 XLAT			AL=（AL）+（BX）	_ _ _ _ _ _ _ _ _
装有效地址指令	LEA	寄 16，存	寄存器=存储器的偏移地址	_ _ _ _ _ _ _ _ _
	LEA（3）	寄 32，存		
装 DS 段值及地址指令	LDS	寄 16，存 32	寄 16=存 32，DS=存 32+2 寄 32=寄 48，DS=存 48+4	_ _ _ _ _ _ _ _ _
	LDS（3）	寄 32，存 48		
装 ES 段值及地址指令	LES	寄 16，存 32	寄 16=存 32，ES=存 32+2 寄 32=存 48，ES=存 48+4	_ _ _ _ _ _ _ _ _
	LES（3）	寄 32，存 48		
装入 SS 段值及地址指令 LSS（3）		寄 16，存 32 寄 32，存 48	寄 16=存 32，SS=存 32+2 寄 32=存 48，SS=存 48+4	_ _ _ _ _ _ _ _ _
装 FS 段值及地址指令 LFS（3）		寄 16，存 32 寄 32，存 48	寄 16=存 32，FS=存 32+2 寄 32=存 48，FS=存 48+4	_ _ _ _ _ _ _ _ _
装 GS 段值及地址指令 LGS（3）		寄 16，存 32 寄 32，存 48	寄 16=存 32，GS=存 32+2 寄 32=存 48，GS=存 48+4	_ _ _ _ _ _ _ _ _
装标志到 AH 指令 LAHF			AH=标志寄存器低字节	_ _ _ _ _ _ _ _ _
送 AH 到标志指令 SAHF			标志寄存器低字节=（AH）	_ _ _ _ _ r r r r r
标志入栈指令 PUSHF			SP=（SP）-2，SS：SP=FLAGS	_ _ _ _ _ _ _ _ _
32 位标志入栈指令 PUSHFD（3）			SP=（SP）-4，SS：SP=EFLAGS	
弹出到 16 位标志寄存器指令 POPF			FLAGS=SS：SP，SP=（SP）+2	r r r r r r r r r
弹出到 32 位标志寄存器指令 POPFD（3）			EFLAGS=SS：SP，SP=（SP）+4	r r r r r r r r r
检查边界指令 BOUND(2)		寄 16，存 32 寄 32，存 16	如果寄存器为 16 位时满足条件：存≤寄≤存+2，为 32 位时满足条件：存≤寄≤存+4，则合法；否则超越边界发中断方式码为“5”的异常，但在实模式下将重复地打印屏幕，在保护模式下有错误处理	
有符号数传送并扩展指令 MOVSX		寄 16，寄 8 寄 16，存 8 寄 16，寄 16 寄 16，存 16 寄 32，寄 8 寄 32，存 8 寄 32，寄 16 寄 32，存 16	将源操作数作为有符号数传送并扩展到目的操作数中。如果目的操作数与源操作数位数相等，则仅实现传送	_ _ _ _ _ _ _ _ _

续表

指 令 名 称	操作数形式	功 能 说 明	标志位 O D I T S Z A P C
无符号数传送并扩展指令 MOVZX	寄 16，寄 8 寄 16，存 8 寄 16，寄 16 寄 16，存 16 寄 32，寄 8 寄 32，存 8 寄 32，寄 16 寄 32，存 16	将源操作数作为无符号数传送并扩展到目的操作数中。如果目的操作数与源操作数位数相等，则仅实现传送	_ _ _ _ _ _ _ _ _
设置堆栈空间指令 ENTER	数 16，数 8	数 16 分配字节空间数，数 8 指出嵌套层数	
二、算术运算类			
加法指令 ADD	寄，寄 寄，存 存，寄 寄，数 存，数	目的操作数=目的操作数+源操作数	x _ _ _ x x x x x
带进位加法指令 ADC	寄，寄 寄，存 存，寄 寄，数 存，数	目的操作数=目的操作数+源操作数+ CF	x _ _ _ x x x x x
加 1 指令 INC	寄/存	操作数=操作数+1	x _ _ _ x x x x _
减法指令 SUB	寄，寄 存，寄 寄，存 寄，数 存，数	目的操作数=目的操作数-源操作数	x _ _ _ x x x x x
带借位减法指令 SBB	寄，寄 寄，存 存，寄 寄，数 存，数	目的操作数=目的操作数-源操作数-CF	x _ _ _ x x x x x
减 1 指令 DEC	寄/存	操作数=操作数-1	x _ _ _ x x x x _
求补指令 NEG	寄 存	操作数=NOT 操作数+1	x _ _ _ x x x x x
比较指令 CMP	寄，寄 寄，存 存，寄 寄，数 存，数	目的操作数-源操作数产生标志位，对源操作数和目的操作数均无影响	x _ _ _ x x x x x
比较传送指令 CMPXCHG（4）	寄，寄 存，寄 寄，存	累加器-目的操作数产生 ZF，如果 ZF=1，则目的操作数=源操作数，否则累加器=目的操作数	x _ _ _ x x x x x

续表

指令名称	操作数形式	功能说明	标志位 ODITSZAPC
比较交换指令 CMPXCHG8B（p）	存 64	EDX-EAX 中的 8 字节值与 8 字节存储器操作数比较，若相等，ZF=1，且 EDX-EAX 送存 64，否则 ZF=0，存 64 送 EDX-EAX	_____x___
无符号乘法指令 MUL	寄 存	字节运算：AX=操作数*AL；字运算：DX-AX=操作数*AX；双字运算：EDX-EAX=操作数*EAX；当积的高位一半为全 0，则 OF=CF=0，否则 OF=CF=1	x___uuuux
有符号乘法指令 IMUL	寄 存	字节运算：AX=操作数*AL；字运算：DX-AX=操作数*AX；双字运算：EDX-EAX=操作数 EAX；如果结果高一半为结果低一半的符号的扩充则 CF=OF=0，否则 CF=OF=1	x___uuuux
有符号乘法指令 IMUL（2）	寄，数 寄，存，数 寄 1，寄 2，数	寄=寄*数 寄=存*数 寄 1=寄 2*数	x___uuuux
无符号除法指令 DIV	寄，存	字节运算：AH，AL=AX÷操作数，其中被除数 AX 为双倍字节长，除后的商在 AL 中，余数在 AH 中；字运算：DX，AX=DX-AX÷操作数，DX-AX 为双字，商在 AX，余数在 DX 中；双字运算：EDX，EAX=EDX-EAX÷操作数，其中被除数 EDX-EAX 为 4 个字长，商在 EAX，余数在 EDX 中，商溢时产生 0 型中断并处理	u___uuuuu
有符号除法指令 IDIV	寄，存	字节运算：AH，AL=AX÷操作数，商-128~127 在 AL 中，余数在 AH 中；字运算：DX，AX=DX-AX÷操作数，DX-AX 为双字，商-32768~32767 在 AX 中，余数在 DX 中；双字运算：EDX，EAX=EDX-EAX÷操作数，商在 EAX 中，余数在 EDX 中。余数和被除数同符号，被除数为除数的双倍长，商溢出时产生 0 型中断并处理	u___uuuuu
有符号字节扩展指令 CBW		把 AL 符号位扩充到 AH 中	_____
有符号扩展指令 CWD		把 AX 的符号位扩展到 DX 中	_____
有符号双字节扩展指令 CDQ（3）		将 EAX 的符号扩展到 EDX 中	_____
非压缩 BCD 数加法调整指令 AAA		如果 AL 的低 4 位大于 9 或 AF=1，则 AL=AL+6，AH=AH+1，CF=AF=1，AL 高 4 位清零，否则，AL 高 4 位清 0，CF=AF=0	u___uuxux
非压缩 BCD 数减法调整指令 AAS		如果 AL 的低 4 位大于 9 或 AF=1，则 AL=AL-6，AH=AH-1，CF=AF=1，AL 高 4 位清零，否则，AL 高 4 位清 0，CF=AF=0	u___uuxux

续表

指 令 名 称	操作数形式	功 能 说 明	标志位 O D I T S Z A P C
非压缩 BCD 数乘法调整指令 AAM		AH，AL=AL÷10 商在 AH 中，余数在 AL 中	u _ _ _ x x u x u
非压缩 BCD 数除法调整指令 AAD		AL=AL+AH10 AH=0	u _ _ _ x x u x u
压缩 BCD 数加法调整指令 DAA		若 AL AND 0FH>9 或 AF=1，则 AL=AL+6 且 AF=1；若 AL≥A0H 或 CF=1；则 AL=AL+60H 且 CF=1，否则 AL 内容不变	u _ _ _ x x x x x
压缩 BCD 数减法调整指令 DAS		若 AL AND 0FH>9 或 AF=1，则 AL=AL−6 且 AF=1；若 AL≥A0H 或 CF=1，则 AL=AL−60H 且 CF=1，否则 AL 内容不变	u _ _ _ x x x x x
三、逻辑运算与移位类			
逻辑乘指令 AND	寄，寄 寄，存 存，寄 寄，数 存，数	目的操作数=目的操作数 AND 源操作数	0 _ _ _ x x u x 0
逻辑加/或指令 OR	寄，寄 寄，存 存，寄 寄，数 存，数	目的操作数=目的操作数 OR 源操作数	0 _ _ _ x x u x 0
异或指令 XOR	寄，寄 寄，存 存，寄 寄，数 存，数	目的操作数=目的操作数⊕源操作数，表示逐位相加，不进位	0 _ _ _ x x u x 0
求反指令 NOT	寄 存	操作数=操作数每存位变反	_ _ _ _ _ _ _ _ _
逻辑比较指令 TEST	寄，寄 寄，存 存，寄 寄，数 存，数	目的操作数 AND 源操作数产生标志位	0 _ _ _ x x u x 0
位测试指令 BT（3）	寄，存 寄，数 存，寄 存，数	源操作数指定内容，目的操作数指定位，被测位的值送 CF	_ _ _ _ _ _ _ x
位测试并求反指令 BTC（3）	寄，存 寄，数 存，寄 存，数	在 BT 指令基础上，被测位取反	_ _ _ _ _ _ _ x
位测试并复位指令 BTR（3）	寄，存 寄，数 存，寄 存，数	在 BT 指令基础上，被测位置 0	_ _ _ _ _ _ _ x

续表

指令名称		操作数形式	功能说明	标志位 ODITSZAPC
位测试及并置位指令 BTS（3）		寄，存 寄，数 存，寄 存，数	在 BT 指令基础上，被测位置 1	_ _ _ _ _ _ _ x
向左位扫描指令 BSF（3）		寄，寄 寄，存	对源操作数从低到高字扫描。首先扫描的“1”的位号送目的操作数，且 ZF=0，若源操作为 0，则目的操作数不确定，且 ZF=1	_ _ _ _ _ x _ _ _
向右位扫描指令 BSR（3）		寄，寄 寄，存	对源操作数从高到低字扫描。首先扫描的“1”的位号送目的操作数，且 ZF=0，若源操作为 0，则目的操作数不确定，且 ZF=1	_ _ _ _ _ x _ _ _
逻辑/算术左移 1 位指令 SHL/SAL		寄，1 存，1	目的操作数内容左移 1 位，低位用 0 补空，高位移到 CF 中，若 CF 不等于操作数最高位移后的值，则 OF=1，否则 OF=0	x _ _ _ x x u x x
逻辑/算术左移指令	SHL/SAL	寄，CL 存，CL	目的操作数内容左移（CL）位或“数”位，低位用 0 补空，即补（CL）个或“数”个 0，CF 为最后一次移位前最高位的值	
	SHL/SAL(2)	寄，数 存，数		
多字节左移指令 SHLD（3）		寄，寄，CL 寄，寄，数 存，寄，CL 存，寄，数	第三操作数为移位次数，CF 与第一操作数联合左移，右边移空的位用第二操作数从左边开始顺序提供	u _ _ _ x x u x x
逻辑右移 1 位指令 SHR		寄，1 存，1	目的操作数内容右移 1 位，高位用 0 补空，CF 为移前最低位的值	x _ _ _ x x u x x
逻辑右移指令	SHR	寄，CL 存，CL	目的操作数内容右移 CL 位或“数”位，高位用 0 补空，CF 为最后 1 次移位前最低位的值	
	SHR（2）	寄，数 存，数		
算术右移 1 位指令 SAR		寄，1 存，1	目的操作数内容右移 1 位，高位用符号位补空，CF 为移前最低位的值	x _ _ _ x x u x x
算术右移指令	SAR	寄，CL 存，CL	目的操作数内容右移 CL 位或“数”位，高位用符号位补空，CF 为最后 1 次移位前最低位的值	
	SAR（2）	寄，数 存，数		
多字节右移指令 SHRD（3）		寄，寄，CL 寄，寄，数 存，寄，CL 存，寄，数	第三操作数为移位次数，将第一操作数右移，右边空的位用第二操作数从右边顺序读出补充，第二操作数不变	u _ _ _ x x u x x

续表

指令名称		操作数形式	功能说明	标志位 ODITSZAPC
循环左移 1 位指令 ROL		寄，1 存，1	目的操作数内容循环左移 1 位，CF 是操作数最高位的原值，若 CF 等于现操作数最高位值，则 OF=1，否则 OF=0	x_ _ _ _ _ _ _ _ _x
循环左移指令	ROL	寄，CL 存，CL	目的操作数内容循环左移 CL 或“数”位，CF 与操作数移位最低位的值相同	
	ROL（2）	寄，数 存，数		
循环右移 1 位指令 ROR		寄，1 存，1	目的操作数内容循环右移 1 位，CF 是操作数最高位的原值，若 CF 不等于操作数最高位值，则 OF=1，否则 OF=0	x_ _ _ _ _ _ _ _ _x
循环右移指令	ROR	寄，CL 存，CL	目的操作数内容右移 CL 或“数”位，CF 为操作数移位最高位移后的值	
	ROR（2）	寄，数 存，数		
进位循环左移 1 位指令 RCL		寄，1 存，1	目的操作数内容连同 CF 循环左移 1 位，如果移位前与移位后的最高 2 位不等，则 OF=1，否则 OF=0	x_ _ _ _ _ _ _ _ _x
进位循环左移指令	RCL	寄，CL 存，CL	目的操作数内容连同 CF 左移 CL 或“数”位	
	RCL（2）	寄，数 存，数		
进位循环右移 1 位指令 RCR		寄，1 存，1	目的操作数内容连同 CF 循环右移 1 位，如果移位前与移位后的最高 2 位不等，则 OF=1，否则 OF=0	x_ _ _ _ _ _ _ _ _x
进位循环右移指令	RCR	寄，CL 存，CL	目的操作数内容连同 CF 循环右移 CL 或“数”位	
	RCR（2）	寄，数 寄，数		
四、串操作类				
字节串传送指令 MOVSB			ES:DI=DS:SI，若 DF=0 则 DI=（DI）+1，SI=（SI）+1；否则 DI=（DI）–1，SI=（SI）–1	_ _ _ _ _ _ _ _ _
字串传送指令 MOVSW			ES:DI=DS:SI，若 DF=0 则 DI=（DI）+2，SI=（SI）+2；否则 DI=（DI）–2，SI=（SI）–2	_ _ _ _ _ _ _ _ _
双字串传送指令 MOVSD（3）			ES:EDI=DS:ESI，若 DF=0 则 DI=（DI）+4，SI=（SI）+4；否则 DI=（DI）–4，SI=（SI）–4	
字节串存指令 STOSB			ES:DI=AL，若 DF=0，则 DI=（DI）+1，否则 DI=（DI）–1	_ _ _ _ _ _ _ _ _
字串存指令 STOSW			ES:DI=AX，若 DF=0，则 DI=（DI）+2，否则 DI=（DI）–2	_ _ _ _ _ _ _ _ _

续表

指 令 名 称	操作数形式	功 能 说 明	标志位 O D I T S Z A P C
双字串存指令 STOSD（3）		ES:DI=EAX，若 DF=0，则 DI=（DI）+4，否则 DI=（DI）−4	
字节串取指令 LODSB		AL=DS:SI，若 DF=0，则 SI=（SI）+1；否则 SI=（SI）−1	_ _ _ _ _ _ _ _ _
字串取指令 LODSW		AX=DS:SI，若 DF=0 则 SI=（SI）+2，否则 SI=（SI）−2	_ _ _ _ _ _ _ _ _
双字串取指令 LODSD（3）		EAX=DS:SI，若 DF=0，则 SI=（SI）+4；否则 SI=（SI）−4	_ _ _ _ _ _ _ _ _
字节串比较指令 CMPSB		DS:SI−ES:DI，若 DF=0，则 SI=（SI）+1，DI=（DI）+1，否则 SI=（SI）−1，DI=（DI）−1	x _ _ _ x x x x x
字串比较指令 CMPSW		DS:SI−ES:DI，若 DF=0 则 SI=（SI）+2，DI=（DI）+2，否则 SI=（SI）−2，DI=（DI）−2	x _ _ _ x x x x x
双字串比较指令 CMPSD（3）		DS:SI−ES:DI，若 DF=0，则 SI=（SI）+4，DI=（DI）+4；否则 SI=（SI）−4，DI=（DI）−4	x _ _ _ x x x x x
字节串扫描指令 SCASB		AL−ES:DI，若 DF=0，则 DI=（DI）+1，否则 DI=（DI）−1	x _ _ _ x x x x x
字串扫描指令 SCASW		AX−ES:DI，若 DF=0，则 DI=（DI）+2，否则 DI=（DI）−2	x _ _ _ x x x x x
字串扫描指令 SCASD（3）		EAX−ES:EDI，若 DF=0，则 DI=（DI）+4，否则 DI=（DI）−4	x _ _ _ x x x x x
重复前缀 REP	（串操作指令）	重复执行串操作指令，重复次数在 CX 中	_ _ _ _ _ _ _ _ _
相等重复前缀 REPE/REPZ	（串操作指令）	重复次数在 CX 中：（1）如果 CX=0 或 ZF=0 则转（3），否则转（2）；（2）执行 1 次串操作指令，CX=CX−1，转（1）；（3）停止重复，执行下面的指令	_ _ _ _ _ _ _ _ _
不等重复前缀 REPNE/REPNZ	（串操作指令）	重复次数在 CX 中：（1）如果 CX=0 或 ZF=1 则转（3），否则转（2）；（2）执行 1 次串操作指令，CX=CX−1，转（1）；（3）停止重复，执行下面的指令	_ _ _ _ _ _ _ _ _
字节串输入指令 INSB（2）		ES:DI=DX 指示的端口中的字节数，若 DF=0，则 DI=（DI）+1；否则 DI=（DI）−1。	_ _ _ _ _ _ _ _ _
字串输入指令 INSW（2）		ES:DI=DX 指示的端口中的字，若 DF=0，则 DI=（DI）+2；否则 DI=（DI）−2	_ _ _ _ _ _ _ _ _
双字串输入指令 INSD（3）		ES:DI=DX 指示的端口中的双字，若 DF=0，则 DI=（DI）+4；否则 DI=（DI）−4	_ _ _ _ _ _ _ _ _
字节串输出指令 OUTSB（2）		把 DS:SI 指示的字节内容送到 DX 指示端口中，若 DF=0 则 SI=（SI）+1，否则 SI=（SI）−1	
字串输出指令 OUTSW（2）		把 DS:SI 指示的字内容送到 DX 指示端口中，若 DF=0 则 SI=（SI）+2，否则 SI=（SI）−2	
双字串输出指令 OUTSD（3）		把 DS:SI 指示的双字内容送到 DX 指示端口中，若 DF=0 则 SI=（SI）+4，否则 SI=（SI）−4	
五、控制转移类（注：IP*表示 IP 或 EIP（对于 80386 以上处理器）；SP*表示 SP 或 ESP（对于 80386 以上处理器））			
近转移指令 JMP	近标号	IP*=OFFSET 近标号	_ _ _ _ _ _ _ _ _

续表

指令名称		操作数形式	功能说明	标志位 ODITSZAPC
远转移指令 JMP		远标号	实模式下 IP*=OFFSET 远标号， CS=SEG 远标号；保护模式通过任务门转移	_ _ _ _ _ _ _ _ _
近间转移指令	JMP	寄 16 存 16	对 80386 以前的 CPU 实现 IP=寄 16/存 16；对 80386 及其后的 CPU 实现 EIP=寄 32/存 32	_ _ _ _ _ _ _ _ _
	JMP（3）	寄 32 存 32		
远间转移指令	JMP	存 32	实模式对 80386 以前的 CPU 实现 IP=存 32，CS=存 32+2；对 80386 及其以后的 CPU 实现 EIP=存 48，CS=存 48+4，保护模式下通过调用门或任务门转移	_ _ _ _ _ _ _ _ _
	JMP（3）	存 48		
等于或全零转移指令	JE/JZ	短标号	若 ZF=1，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JE/JZ （3）	近标号		
不等于转移或非零转移指令	JNE/JNZ	短标号	若 ZF=0，则 IP*=OFFSET 标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
	JNE/JNZ（3）	近标号		
负号转移指令	JS	短标号	若 SF=1，则 IP*=OFFSET 标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
	JS（3）	近标号		
正号转移指令	JNS	短标号	若 SF=0，则 IP*=OFFSET 标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
	JNS（3）	近标号		
溢出转移指令	JO	短标号	若 OF=1，则 IP*=OFFSET 标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
	JO（3 ）	近标号		
无溢出转移指令	JNO	短标号	若 OF=0，则 IP*=OFFSET 标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
	JNO（3）	近标号		
偶性奇偶转移指令	JP/JPE	短标号	若 PF=1，则 IP*=OFFSET 标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
	JP/JPE（3）	近标号		
奇性奇偶转移指令	JNP/JPO	短标号	若 PF=0，则 IP*=OFFSET 标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
	JNP/JPO（3）	近标号		
低于转移或进位转移指令	JB/JC/JNAE	短标号	若 CF=1，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JB/JC/ JNAE （3）	近标号		
高于等于转移或无进位转移指令	JAE/JNB/ JNC	短标号	若 CF=0，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JAE/JNB/ JNC（3）	近标号		
低于等于转移指令	JBE/JNA	短标号	若 CF 或 ZF=1，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JBE/JNA（3）	近标号		

续表

指令名称		操作数形式	功能说明	标志位 ODITSZAPC
高于转移指令	JA/JNBE	短标号	若 CF=ZF=0，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JA/JNBE （3）	近标号		
小于转移指令	JL/JNGE	短标号	若 ZF=0 且 SF≠OF，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JL/JNGE（3）	近标号		
大于等于转移指令	JGE/JNL	短标号	若 ZF=1 或 SF=OF，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JGE/JNL（3）	近标号		
小于等于转移指令	JLE/JNG	短标号	若 ZF=1 或 SF≠OF，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JLE/JNG（3）	近标号		
大于转移指令	JG/JNLE	短标号	若 ZF=0 且 SF=OF，则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
	JG/JNLE（3）	近标号		
计数零转移指令 JCXZ 或 JEXCZ（3）		短标号	若 CX=0（JCXZ）或者 ECX=0（JECXZ），则 IP*=OFFSET 标号；否则 IP*不变	_ _ _ _ _ _ _ _ _
计数循环指令 LOOP		短标号	CX=（CX）-1，若 CX≠0，则 IP*=OFFSET 短标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
等计数循环指令 LOOPE/零计数循环指令 LOOPZ		短标号	CX=（CX）-1，若 CX≠0，且 ZF=1，则 IP*=OFFSET 短标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
不等计数循环 LOOPNE/非零计数循环指令 LOOPNZ			CX=（CX）-1，若 CX≠0，且（ZF）=0，则 IP*=OFFSET 短标号，否则 IP*不变	_ _ _ _ _ _ _ _ _
近直接调用指令 CALL		近标号	16 位地址：SP=（SP）-2 SS:SP=IP，IP=OFFSET，近标号；32 位地址：SP*=（SP*）-4，SS:SP*=IP，EIP*=OFFSET，近标号，对 32 位栈段保护模式下特权检验，不合法发 0DH 异常	_ _ _ _ _ _ _ _ _
近间调用指令 CALL		存 16 寄 16 存 32 寄 32	16 位地址：SP=（SP）-2，SS:SP=IP，IP=操作数 16；32 位地址：SP*=(SP*)-4 SS:SP*=IP*，IP*=操作数 32；对 32 位栈段保护模式下特权检验，不合法发 0DH 异常	_ _ _ _ _ _ _ _ _
远直接调用指令 CALL		远标号	16 位地址：SP=（SP）-2，SS:SP=CS，SP=（SP）-2 SS:SP=IP，CS=SEG，远标号，IP=OFFSET 远标号；32 位地址；SP*=（SP*）-4，SS:SP*=CS，SP*=(SP*)-4，SS:SP*=IP*CS=SEG 远标号，IP*=OFFSET 远标号	_ _ _ _ _ _ _ _ _
远间调用指令 CALL		存 32 存 48（3）	16 位地址：SP=（SP）-2，SS:SP=CS SP=（SP）-2，SS:SP=IP，CS=存 32+2，IP=存 32；32 位地址：SP*=（SP*）-4，SS:SP*=CS 扩充为 32 位，SP*=（SP*）-4，SS:SP*=IP*，CS=存 48+4，IP*=存 48	_ _ _ _ _ _ _ _ _

续表

指令名称	操作数形式	功能说明	标志位 ODITSZAPC
远返回指令 RETF 或 RET（远过程中的语句）	空 数 16 数 32（3）	16 位操作数：IP=SS:SP，SP=（SP）+2 或 SP=（SP）+2+数 16，CS=SS:SP，SP=（SP）+2 或 SP=（SP）+2+数 16；32 位操作数：IP*=SS:SP*，SP*=（SP*）+4 或 SP*=（SP*）+4+数，CS=SS:SP*，SP*=（SP*）+4 或 SP*=（SP*）+4+数	_ _ _ _ _ _ _ _ _
近返回指令 RETN 或 RET（近过程中的语句）	空 数 16 数 32（3）	16 位操作数：IP=SS:SP，SP=（SP）+2 或 SP=（SP）+2+数 16；32 位操作数：IP*=SS:SP*，SP*=（SP*）+4 或 SP*=（SP*）+4+数	_ _ _ _ _ _ _ _ _
中断调用指令 INT	数	16 位地址：SP=（SP）−2，SS:SP=FLAGS，IF=TF=0，SP=（SP）−2，SS:SP=CS，SP=（SP）−2，SS:SP=IP，实模式:SS=数 4+2，IP=数 4；32 位地址：SP*=（SP*）−4，SS:SP*=EFLAGS，IF=TF=0，SP*=（SP*）−4，SS:SP*=CS 扩充为 32 位，SP*=（SP*）−4，SS:SP*=IP*。保护模式由中断方式码经过中断门描述转中断处理程序	_ _ 0 0 _ _ _ _ _
溢出中断调用指令 INT0		当 OF=0 时无操作。当 OF=1 时，16 位地址：保留标志数寄存器 SP=(SP)−2，SS:SP=FLAGS 对中，IF=TF=0，保留断点 SP=（SP）−2，SS:SP=CS，SP=IP，实模式实现转移 CS=44+2，IP=44；32 位地址：SP*=（SP*）−4 SS:SP*=EFLAGS，IF=TF=0，SP*=（SP*）−4，SS:SP*=CS 扩充为 32 位，SP*=（SP*）−4，SS:SP*=IP*保护模式由中断方式码“4”经中断门和任务描述子相应中断处理程序	_ _ 0 0 _ _ _ _ _
中断返回指令 IRET		实模式下有下列功能：恢复断点，IP*=SP*，SP*=（SP*）+2，CS=SS:SP*，SP*=（SP*）+2 保护模式下它启动任务转换返回，要经过特权检查	r r r r r r r r r
六、处理器控制类			
清进位标志 CLC		CF=0	_ _ _ _ _ _ _ 0
置进位标志指令 STC		CF=1	_ _ _ _ _ _ _ 1
进位标志位求反 CMC		对进位标志位 CF 求反	_ _ _ _ _ _ _ x
清方向标志 CLD		DF=0	_ 0 _ _ _ _ _ _
置方向标志指令 STD		DF=1	_ 1 _ _ _ _ _ _
清中断标志指令 CLI		IF=0	_ _ 0 _ _ _ _ _
置中断标志指令 STI		IF=1	_ _ 1 _ _ _ _ _
处理器特征识别指令 CPUID		根据 EAX 中的参数，将处理器的说明信息送 EAX，特征标志字送 EDX	
读时间标记计数器指令 RDTSC		将 Pentium 中的 64 位时间标志计数器的字位送 EDX，低 32 位送 EAX	
读模型专用寄存器指令 RDMSR		将 E C X 所指定的模型专用寄存器的内容送 EDX、EAX	

续表

指令名称	操作数形式	功能说明	标志位 ODITSZAPC
写模型专用寄存器指令 WRMSP		将 EDX、EAX 的内容送到由 ECX 指定的专用寄存器	
片内 Cache 无效指令 INVD		该指令用于将 CPU 内部 Cache 的内容无效，不写回主存	
写回并使 Cache 无效指令 WBINVD		该指令用于将 CPU 内部 Cache 的内容无效，写回主存	
使 TLB 无效指令 INVLPG	存	页式管理机构内的高速缓冲器 TLB 中的某项作废	
空操作指令 NOP		CPU 空转 3 个时间片	_ _ _ _ _ _ _ _ _
暂停指令 HLT		使处理器暂停执行	_ _ _ _ _ _ _ _ _
等待指令 WAIT		使 CPU 等待输入端为有效。执行等待时能响应中断，等待结束时返回下条语句	_ _ _ _ _ _ _ _ _
处理器转移指令 ESC	数，寄 数，存	ESC 指令向协处理（8087，80287，80387）提供一条可执行的指令及相应的操作数，ESC 的第一个操作数为 6 位常数，它表示协处理器指令的操作码，第二个操作数则是协处理指令的操作数	_ _ _ _ _ _ _ _ _
封锁总线指令 LOCK	INS，MOV，OUTS，XCHG，ADD，ADC，OR，AND，SBB，XOR，INC，NEG，XCHG…	执行操作数处指令时对总线进行封锁。如果超出左列指令范围，则产生非法操作码异常。注：标志位变化与相应指令相同	_ _ _ _ _ _ _ _ _
七、杂类			
高于设置指令 SETA/SETNBE（3）	寄 8 存 8	若 CF=0 和 ZF=0 即高于或不低于等于，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
高于等于设置指令 SETAE/SETNB（3）	寄 8 存 8	若 CF=0 即高于等于或不等于，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
低于设置指令 SETB/SETNAE（3）	寄 8 存 8	若 CF=1 即低于或不高于等于，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
低于等于设置指令 SETBE/SETNA（3）	寄 8 存 8	若 CF=1 或 ZF=1 即低于等于或不高于，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
等于设置指令 SETE/SETZ（3）	寄 8 存 8	若 ZF=1 即等于或为 0，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
不等于设置指令 SETNE/SETNZ（3）	寄 8 存 8	若 ZF=0 即不等于或不为 0，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
大于设置指令 SETG/SETNLE（3）	寄 8 存 8	若 ZF=0 或 SF=OF 即大于或不小于等于，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
大于等于设置指令 SETGE/SETNL（3）	寄 8 存 8	若 SF=OF 即大于等于或不小于，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
小于设置指令 SETL/SETNGE（3）	寄 8 存 8	若 SF≠OF 即小于或不大于等于，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
小于等于设置指令 SETLE/SETNG（3）	寄 8 存 8	若 ZF=1 或 SF≠OF 即小等于于或不大于，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _

续表

指 令 名 称	操作数形式	功 能 说 明	标志位 O D I T S Z A P C
无溢出设置指令 SETNO（3）	寄 8 存 8	若 OF=0 即无溢出，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
溢出设置指令 SETO（3）	寄 8 存 8	若 OF=1 即溢出，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
正号设置指令 SETNS（3）	寄 8 存 8	若 SF=0 即为正号，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
负号设置指令 SETS（3）	寄 8 存 8	若 SF=1 即为负号，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
奇校设置指令 SETNP/SETPO（3）	寄 8 存 8	若 PF=0 即奇校验，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
偶校设置指令 SETP/SETPE（3）	寄 8 存 8	若 PF=1 即偶校验，则操作数=1，否则操作数=0	_ _ _ _ _ _ _ _ _
存全局描述符表寄存器指令 SGDT（2）	存 40 存 48（3）	操作数=GDTR，限长在小地址中占两个字节，基址在后面三个字节中最后一个字节对 80386 及其以后 CPU 为全 0；对 80286CPU 为无定义	_ _ _ _ _ _ _ _ _

附录D　MASM 5.0 宏汇编程序出错信息

汇编程序在对源程序的汇编过程中，若检查出某语句有语法错误，随时在屏幕上给出出错信息。如操作人员指定了列表文件名（即.LST），汇编程序亦将在列表文件中出错行的下面给出出错信息，以便操作人员即时查找错误，给予更正。MASM 5.0 出错信息格式如下：

源程序文件行：WARNING/ERROR 错误信息码：错误描述信息

其中，错误信息码由五个字符组成。第一个是字母 A，表示汇编语言程序出错；接着有一个数字指明出错类别：‘2’为严重错误，‘4’为严肃警告，‘5’为建议性警告，最后三位为错误编号。

错 误 编 号	错 误 描 述
0	Block　nesting error 嵌套出错。嵌套的过程、段、结构、宏指令或重复块等非正常结束。例如在嵌套语句中有外层的结束语句，而无内层的结束语句
1	Extra characters on line 一语句行有多余字符，可能是语句中给出的参数太多
2	Internal error - Register already defined 这是一个内部错误。如出现该错误，请记下发生错误的条件，并使用 Product Assistance Request 表与 Mi-crosoft 公司联系
3	Unknown type specifer 未知的类型说明符。例如类型字符拼错，把 BYTE 写成 BIT，NEAR 写成 NAER 等
4	Redefinition of symbol 符号重定义。同一标识符在两个位置上定义。在汇编第一遍扫描时，在这个标识符的第二个定义位置上给出这个错误
5	Symbol is　multidefined 符号多重定义。同一标识符在两个位置上定义。在汇编第二遍扫描时，每当遇到这个标识符都给出这个错误
6	Phase error between passes 两次扫描间的遍错。一个标号在二次扫描时得到不同的地址值，就会给出这种错误。若在启动 MASM 时使用/D 任选项，产生第一遍扫描的列表文件，它可帮助你查找这种错误
7	Already had ELSE clause 已有 ELSE 语句。在一个条件块里使用多于一个的 ELSE 语句
8	Must be in conditional block 没有在条件块里。通常是有 ENDIF 或 ELSE 语句，而无 IF 语句
9	Symbol not defined 符号未定义。在程序中引用了未定义的标识符
10	Syntax error 语法错误。不是汇编程序所能识别的一个语句
11	Type illegal in context 指定非法类型。例如对一个过程指定 BYTE 类型，而不是 NEAR 或 FAR.
12	Group name must be unique 组名应是唯一的。作为组名的符号做为其他符号使用

续表

错 误 编 号	错 误 描 述
13	Must be declared during pass 1 必须在第一遍扫描期间定义。在第一遍扫描期间，如一个符号在未定义前就引用，就会出现这种错误。例如 HEX1 未定义前就出现 IF 的 HEX1 语句
14	Illegal public declaration 一个标识符被非法的指定为 PUBLIC 类型
15	Symbol already different kind 重新定义一个符号为不同种类符号。例如一个段名重新被当作变量名定义使用
16	Reserved word used as symbol 把汇编语言规定的保留字做标识符使用
17	Forward reference illegal 非法的前向引用。在第一遍扫描期间，引用一个未定义符号。例如： DB CUNT DUP（0） CUNT EQU 10H 对调上述二语句的顺序即为合法。并非任何前向引用都是错误的
18	Operand must be register 操作数位置上应是寄存器，但出现了标识符
19	Wrong type of register 使用的寄存器类型出错。如“LEA AL，VAR”就属于这种错误
20	Operand must be segment or group 应该给出一个段名或组名（group）。例如 ASSUME 语句中应为某段寄存器指定一个段名或组名，而不应是别的标号或变量名等
21	Symbol has no segment 不知道标识符的段属性
22	Operand must be type specifier 操作数应给出类型说明符，如 NEAR、FAR、BYTE、WORD 等
23	Symbol already defined locally 已被指定为内部（local）的标识符，企图在 EXTRN 语句中又定义外部标识符
24	Segment parameters are changed 段参数被改变。如同一标识符定义在不同段内
25	Improper align/combine type 段定义时的定位类型/组合类型使用出错
26	Reference to multidefined symbol 指令引用了多重定义的标识符
27	Operand expected 需要一个操作数，但只有操作符，如“MOV BX，OFFSET”
28	Operator expected 需要一个操作符，但只有操作数
29	Division by 0 or overflow 除以 0 或溢出
30	Negative shift count\ 运算符 SHL 或 SHR 的移位表达式值为负数
31	Operand type must match 操作数类型不匹配。双操作数指令的两个操作数长度不一致，一个是字节，一个是字
32	Illegal use of external 外部符号使用出错

续表

错 误 编 号	错 误 描 述
33	Must be record field name 应为记录字段名。在记录字段名位置上出现另外的符号
34	Must be record name of field name 应为记录名或记录字段名。在记录名或记录字段名位置上出现另外的符号
35	Operand must have size 应指明操作数的长度（如 BYTE、WORD 等）。通常使用 PTR 运算即可改正错误
36	Must be variable，label or constant 应该是变量名、标号或常数的位置上出现了其他信息
37	Must be structure field name 应为结构字段名。在结构字段名位置上出现了另外的符号
38	Left operand must segment 操作数的左边应是段的信息。如设 DA1、DA2 均是变量名，下列语句就是错误的：“MOV AX，DA1:DA2”。在 DA1 位置上应使用某段寄存器名
39	One operand must be constant 操作数必须是常数。例如一个表达式中用‘+’运算符把两个变量名相加出错。 而用‘+’运算必须有一个常数
40	Operand must be in same segment or one constant “－“运算符用错。例如“MOV　AL，－VAR”，其中 VAR 是变量名，应有一常数参加运算。又如两个不同段的变量名相减出错
41	Normal type operand expected 要求给出一个正常的操作数。例如在变量名的位置上出现了另外的符号或信息
42	Constant expected 要求给出一个常数。例如给出一个不是常数的操作数或表达式
43	Operand must have segment 运算符 SEG 用错。如 SEG 后跟一个常数，而常数没有段属性
44	Must be associated with data 在必须与数据段有关的位置上出现了代码段有关的项。例如：“MOV AX，LENGTH CS：VAR”。其中 VAR 是数据段中的变量名
45	Must be associated with code 在必须与代码段有关的位置上出现了数据段有关的项
46	Multiple base registers 同时使用了多个基址寄存器。例如：“MOV AX ，[BX][BP]”
47	Multiple index registers 同时使用了多个变址寄存器。例如：“MOV AX ，[SI][DI]”
48	Must be index or base register 指令仅要求使用基址寄存器或变址寄存器，而不能用其他寄存器。例如：“MOV AX ，[SI+CX]”
49	Illegal use of register 非法使用寄存器出错
50	Value is out of range 数值太大，超过允许值。例如：“MOV AL，100H”
51	Operand not in current CS ASSUME segment 操作数不在当前代码段内。通常指转移指令的目标地址不在当前 CS 段内
52	Improper operand type 操作数类型使用不当。例如：“MOV VAR1，VAR2”。两个操作数均为存储器操作数，不能汇编出目标代码

续表

错 误 编 号	错 误 描 述
53	Jump out of rang by %ld byte（s） 条件转移指令跳转范围超过-128～+127 个字节。出错信息同时给出超过的字节数
54	Index displacement must be constant 变址寻址的位移量必须是常数
55	Illegal register value 非法的寄存器值。目标代码中表达寄存器的值超过“7”
56	Immediate mode illegal 不允许使用立即数寻址方式。例如：“MOV DS，CODE”，其中 CODE 是段名，不能把段名作立即数传送给段寄存器 DS
57	Illegal size for operand 使用的操作数大小（字节数）出错。例如，使用双字（32 位）的存储器操作数
58	Byte register illegal 要求用字寄存器的指令使用了字节寄存器。如 PUSH，POP 指令的操作数寄存器就必须是字寄存器（16 位）
59	Illegal use of CS register 指令中错误地使用段寄存器 CS。如：“MOV CS，AX”，CS 不能做目的操作数
60	Must be accumulator register 要求用 AX 或 AL 的位置上出现了其他寄存器。如 IN，OUT 指令必须使用累加器 AX 或 AL
61	Improper use of segment register 不允许用段寄存器的位置上使用了段寄存器。如“SHL DS，1”指令
62	Missing or unreachable CS 试图跳转去执行一个 CS 达不到的标号。通常是指缺少 ASSUME 语句中 CS 与代码段相关联
63	Operand combination illegal 双操作数指令中两个操作数组合出错
64	Near JMP/CALL to different CS 试图用 NEAR 属性的转移指令跳转到不在当前段的一个地址
65	Label cannot have segment override 段前缀使用出错
66	Must have instruction after prefix 在重复前缀 REP，REPE，REPNE 后面必须有指令
67	Cannot override ES for destination 串操作指令中目的操作数不能用其他段寄存器替代 ES
68	Cannot address with segment register 指令中寻找一个操作数，但 ASSUME 语句中未指明哪个段寄存器与该操作数所在段有关联
69	Must be in segment block 指令语句没有在段内
70	Cannot use EVEN or ALIGN with byte alignment 在段定义伪指令的定位类型中选用 BYTE，这时不能使用 EVEN 或 ALIGN 伪指令
71	Forward needs override or FAR 转移指令的目标没有在源程序中说明为 FAR 属性，可用 PTR 指定
72	Illegal value for DUP count 操作符 DUP 前的重复次数是非法的（如负数）或未定义
73	Symbol is already external 在模块内试图定义的符号，它已在外部符号伪指令中说明
74	DUP nesting too deep 操作数 DUP 的嵌套太深

续表

错 误 编 号	错 误 描 述
75	Illegal use of undefined operand（?） 不定操作符‘？’使用不当。例如“DB 10H DUP（?+2）”
76	Too many value for struc or record initialization 在定义结构变量或记录变量时，初始值太多
77	Angle brackets required around initialized list 定义结构变量时，初始值未用尖括号（<>）括起来
78	Directive illegal structure 在结构定义中的伪指令语句使用不当。结构定义中伪指令语句仅二种：分号（;）开始的注释语句和用 DB、DW 等数据定义伪指令语句
79	Override with DUP illegal 在结构变量初始值表中使用 DUP 操作符出错
80	Field cannot be overridden 在定义结构变量语句中试图对一个不允许修改的字段设置初值
81	Override is of wrong type 在定义结构变量语句中设置初值时类型出错
82	Circular chain of EQU aliases 用等值语句定义的符号名，最后又返回指向它自己。如： A EQU B B EQU A
83	Cannot emulate cooprocessor opcode 仿真器不能支持的 8087 协处理器操作码
84	End of file，no END directive 源程序文件无 END 文件
85	Data emitted with no segment 数据语句没有在段内

附录E　DEBUG命令表

说明：表中地址有段值；偏移值、段寄存器名，偏移值；偏移地址三种形式。仅有偏移地址时，如果不指明，则段值均为 DS 中的内容。

名　称	命 令 格 式	功 能 说 明
A	A[地址]	汇编半源程序到指定地址中，仅有偏移地址时，段值在 CS 中
C	C 地址 1 L 长度 地址 2	比较地址 1 与地址 2 开始的“长度”个字节内容，如果发现不相等，则显示出地址与内容
D	D 地址 L 长度或 D [地址 末址]	显示指定始址及范围的内容，按字节显示十六进制及字符数。如无参数，则按 DS:0 开始显示 128 字节内容，可连续使用 D 接着显示
E	E 地址[数据表]	将数据表中十六进制字节或字符串写入该地址开始的字节中。如无数据表，则先显示该字节的内容，然后等待输入 1 个字节的内容
F	F 地址 L 长度数据表或 F 始址末址数据表	将数据表中字节数或字符串填入地址所指范围中，如果数据不够长，则重复使用；如果数据过长，则截断
G	G[=始址][地址]…	从指定始址带断点执行程序。若无始址，则从 CS:IP 开始执行；若无断点，则连续执行。可写 10 个断点[地址]。仅偏移地址时，段值在 CS 中
H	H 数 1 数 2	显示出这两个十六进制数的和与差
I	I 端口号	读出并显示该端口内容
L	L[地址[驱动器号]扇区号扇区数]	从指定设备指定扇区号装入“扇区数”个扇区信息到指定内存中。如果只有偏移地址，则默认 CS 的段值；如果无驱动器号（0 为 A 盘，1 为 B 盘），则为默认盘；如果只有地址或无地址，则将 CS:80H 参数区处的文件装入内存指定地址或 CS:100H 处。该文件可用 N 命令指定
M	M 地址 1 L 长度地址 2 或 M 地址 1 地址 2 地址 3	将地址 1“长度”的内容传送到地址 2 中。或将开始地址 1 至末地址 2 的内容传送到地址为始址的内存中
O	O 端口号值	将指定值输出到指定端口中
P	P[=地址][数]	从指定地址开始追踪执行“数”条语句，每条语句显示 1 次现场内容（主要是寄存器内容及下一次要执行的语句），但子程序相当于 1 条语句，连续执行返回才显示。仅偏移地址时，段值在 CS 中；地址省略，从 CS：IP 指示的地址开始“数”省略时，只执行 1 条语句
Q	Q	退出 DEBUG 系统
R	R[寄存器名]	不带参数时，显示所有寄存器内容的参数。若要对所指寄存器写入，先显示该寄存器内容，然后再作修改。如果只按回车键，则原值不变。标志寄存器可以整个写入，用“F”作 FLAGS 的名字，也可以单个标志写入。以下为所用标志的名字及其值的符号（不能写 0 或 1，只能用此符号）： 标志位名字 OF DF IF SF ZF AF PF CF 设置（置 1）符号 OV DN EI NG ZR AC PE CY 清除（置 0）符号 NV UP DI PL NZ NA PO NC

续表

名　称	命 令 格 式	功 能 说 明
S	S 地址 L 长度数据表或 S 地址 1 地址 2 数据表	在指定地址范围检索数据表中的数据。将找到的数据地址全部显示出，否则显示找不到的信息。数据表中的数据可以是十六进制字节数或字符串。必须全部相等才算找到。地址 2 为末址
T	T[=地址][数]	从指定地址开始追踪执行“数”条语句，每执行 1 条语句显示 1 次现场内容（主要是寄存器内容及下一次要执行的语句），地址省略时则从 CS:IP 指示的地址开始，无“数”则执行 1 条语句
U	U[地址]或 U[地址 1 地址 2]或 U[地址 L 长度]	反汇编，即把内存字节内容按指令解释用半源程序显示出来。无地址时从 CS:IP 指示的地址开始；仅有偏移地址时，默认 CS 中的段值。无地址范围时，1 次显示 32 个字节内容。可以连续使用 U（不用给地址）连续输出，且不破坏原 IP 的内容
W	W 地址[驱动器号]扇区号 扇区数或 W[地址]	将指定地址内容写入到指定扇区中，无驱动器号（1 为 A 盘、2 为 B 盘等）时则为默认的驱动器。地址只有偏移值时，用 CS 中的段值。如果无参数或只有地址参数，则按 N 定义的文件存盘。最好在用 W 前（中间不要有别的操作）使用 N 定义该文件
N	N [驱动器名：][路径名]文件名[.扩展名]	定义文件建立文件控制块，以供 L 与 W 命令使用，所指定文件的说明存放在 CS:80H 参数区（程序前缀区）PSP 中

附录F　BIOS和MS-DOS功能调用

附表 F.1　BIOS 功能调用

INT	AH	功　能	入 口 参 数		出 口 参 数
10	0	设置显示方式	AL	=00 40×25 黑白文本，16 级灰度	
				=01　40×25 16 色文本	
				=02　80×25 黑白文本，16 级灰度	
				=03　80×25 黑白文本，16 色文本	
				=04　320×200　4 色图形	
				=05　320×200　黑白图形　6 级灰度	
				=06　640×200　黑白图形	
				=07　80×25 黑白文本	
				=08　160×200　16 色图形（MCGA）	
				=09　320×200　16 色图形（MCGA）	
				=0A　640×200　4 色图形（MCGA）	
				=0D　320×200　16 色图形（EGA/VGA）	
				=0E　640×200　16 色图形（EGA/VGA）	
				=0F　640×350　单色图形（EGA/VGA）	
				=10　640×350　16 色图形（EGA/VGA）	
				=11　640×480　黑白图形（VGA）	
				=12　640×480　16 色图形（VGA）	
				=13　320×200　256 色图形（VGA）	
10	1	置光标类型	（CH）$_{0-3}$=光标起始行		
			（CL）$_{0-3}$=光标结束行		
10	2	置光标位置	BH=页号		
			DH/DL=行/列		
10	3	读光标位置	BH=页号		CH_{0-3}=光标起始行
					CL_{0-3}=光标结束行
					DH/DL=行/列
10	4	读光笔位置			AX=0 光笔未触发
					=1 光笔触发
					CH/BX=像素行/列
					DH/DL=字符行/列
10	5	置当前显示页	AL=页号		
10	6	屏幕初始化	AL=0 初始化窗口		
		或上卷	AL=上卷行数		
			BH=卷入行属性		
			CH/CL=左上角行/列号		
			DH/DL=右上角行/列		
10	7	屏幕初始化	AL=0 初始化窗口		
		或下卷	AL=下卷行数		
			BH=卷入行属性		
			CH/CL=左上角行/列号		
			DH/DL=右上角行/列		

续表

INT	AH	功　能	入口参数	出口参数
10	8	读光标位置的	BH=显示页	AH/AL=字符/属性
		字符和属性		
10	9	在光标位置显示	BH=显示页	
		字符和属性	AL/BL=字符/属性	
			CX=字符重复次数	
10	A	在光标位置	BH=显示页	
		显示字符	AL=字符	
			CX=字符重复次数	
10	B	置彩色调色板	BH=彩色调色板 ID	
			BL=和 ID 配套使用颜色	
10	C	写像素	AL=颜色值	
			BH=页号	
			DX/CX=像素行/列	
10	D	读像素	BH=页号	
			DX/CX=像素行/列	AL=像素的颜色值
10	E	显示字符	AL=字符	
		（光标前移）	BH=页号	
			BL=前景色	
10	F	取当前显示方式		BH=页号
				AH=字符列数
				AL=显示方式
10	10	置调色板寄存器	AL=0,BL=调色板号,BH=颜色值	
		（EGA/VGA）		
10	11	装入字符发生器	AL=0～4 全部或部分装入字符点阵集	
		（EGA/VGA）	AL=20～24 置图形方式显示字符集	
			AL=30 读当前字符信息	ES:BP=字符集位置
INT	AH	功能	入口参数	出口参数
10	12	返回当前适配器	BL=10H(子功能)	BH=0 单色方式
		设置的信息		=1 彩色方式
		（EGA/VGA）		BL=VRAM 容量
				（0=64K,1=128K,…）
				CH=特征位设置
				CL=EGA 的开关设置
10	13	显示字符串	ES:BP=字符串地址	
			AL=写方式（0～3）	
			CX=字符串长度	
			DH/DL=起始行/列	
			BH/BL=页号/属性	
11		取设备信息		AX=返回值（位映像）
				0=设备未安装
				1=设备未安装
12		取内存容量		AX=字节数（KB）
13	0	磁盘复位	DL=驱动器号	失败：AH=错误码
			（00，01 为软盘，80ｈ，81ｈ，…为硬盘）	
13	1	读磁盘驱动器		AH=状态字节
		状态		

续表

INT	AH	功　能	入口参数	出口参数
13	2	读磁盘扇区	AL=扇区数	读成功
			（CL）6,7（CH）0～7=磁道号	AH=0
			（CL）0～5=扇区号	AL=读取的记区数
			DH/DL=磁头号/驱动器号	读失败
			ES:BX=数据缓冲区地址	AH=错误码
13	3	写磁盘扇区	同上	写成功
				AH=0
				AL=写入的扇区数
				写失败
13	4	检验磁盘扇区	AL=扇区数	AH=错误码
			（CL）6,7（CH）0～7=磁道号	成功：ＡＨ=0
			（CL）0～5=扇区号	AL=检验的扇区数
			DH/DL=磁头号/驱动器号	失败：AH=错误码
13	5	格式化盘磁道	AL=扇区数	
			（CL）6,7（CH）0～7=磁道号	
			（CL）0～5=扇区号	
			DH/DL=磁头号/驱动器号	
			ES:BX=格式化参数表指针	
INT	AH	功能	入口参数	出口参数
14	0	初始化串行口	AL=初始化参数	AH=通信口状态
			ＤＸ=串行口号	AL=调制解调器状态
14	1	向通信口写字符	AL=字符	写成功：（AH）7=0
			ＤＸ=通信口号	写失败：（AH）7=1
				(AH)0～6=通信口状态
14	2	从通信口读字符	ＤＸ=通信口号	读成功：（AH）7=0
				(AL)=字符
				读失败：（AH）7=1
14	3	取通信口状态	ＤＸ=通信口号	AH=通信口状态
				AL=调制解调器状态
14	4	初始化扩展 com		
14	5	扩展ＣＯＭ控制		
15	0	启动盒式磁带机		
15	1	停止盒式磁带机		
15	2	磁带分块读	ES:BX=数据传输区地址	AH=状态字节
		ＣＸ=字节数		=00 读成功
				=01　冗余检验错
				=02　无数据传输
				=04　无引导
				=80　非法命令
15	3	磁带分块读	DS:BX=数据传输入区地址	AH=状态字节
		ＣＸ=字节数		（同上）
16	0	从键盘读字符		AL=字符码
				AH=扫描码
16	1	取键盘缓冲区状态		ZF=0 AL=字符码
				AH=扫描码
				ZF=1 缓冲区无按键等待

续表

INT	AH	功　能	入口参数	出口参数
16	2	取键盘标志字节		AL=键盘标志字节
	3	设置击键重复速率	AL=5	
			BH=重复延迟（0=250ms，1=500ms，2=750ms，3=1000ms）	
			BL=重复速率(0 最快~1F 最慢)	
	5	按键送入缓冲区	CH=扫描码	如果缓冲区已满
		将键盘字符和其对应的扫描	CL=字符	设置进位标志位
		码送入键盘缓冲区		AL=1
	10	等待输入字符及其扫描码		AH=扫描码
		（注：同 0H，用于老式键盘）		AL=字符的 ASCII 码
INT	AH	功能	入口参数	出口参数
	11	检查按键缓冲区		如有按键等待
		（注：同 01H，用于老式键盘）		AH=扫描码
				AL=字符的 ASCII 码
				如无，设置零标志位
	12	获取键盘标志		AX=键盘标志
		（注：同 02H，用于老式键盘）		
17	0	打印字符	AL=字符	AH=打印机状态字节
		回送状态字节	DX=打印机号	
17	1	初始化打印机	DX=打印机号	AH=打印机状态字节
		回送状态字节		
17	2	取打印机状态	DX=打印机号	AH=打印机状态字节
18		ROM BASIC 语言		
19		引导装入程序		
1A	0	读时钟		CH:CL=时：分
				DH:DL=秒：1/100 秒
1A	1	置时钟	CH:CL=时：分　　DH:DL=秒：1/100 秒	
1A	6	置报警时间	CH:CL=时：分	
			DH:DL=秒：1/100 秒	
1A	7	清除报警		
33	00	鼠标复位	AL=00	BX=鼠标的键数
33	00	显示鼠标光标	AL=01	显示鼠标光标
33	00	隐藏鼠标光标	AL=02	隐藏鼠标光标
33	00	读鼠标状态	AL=03	BX=键状态
				CX/DX=鼠标水平/垂直位置
33	00	设置鼠标位置	AL=04	
			CX/DX=鼠标水平/垂直位置	
33	00	设置图形光标	AL=09	安装了新的图形光标
			BX/CX=鼠标水平/垂直中心	
			ES:DX=16×16 光标映像地址	
33	00	设置文本光标	AL=0A	设置的文本光标
			BX=光标类型	
			CX=像素位掩码或起始的扫描线	
			DX=光标掩码或结束的扫描线	
33	00	读移动计数器	AL=0B	CX/DX=鼠标水平/垂直距离
33	00	设置中断子程序	AL=0C　　CX=中断掩码	
			ES:DX=中断服务程序的地址	

表 F.2　常用 PC 中断号列表

中断号	功 能 说 明
0	除法错误。CUP 产生。当试图除零时发生
1	单步。CPU 产生。当陷阱标志设置时发生
2	不可屏蔽中断。外部硬件。当发生内存错误时产生
3	断点：CPU 产生。当执行机器码 0CCh（INT3）时发生
4	除法溢出。CPU 产生。溢出标志设置的条件下执行 INTO 指令时发生
5	打印屏幕。执行 INT5 或按下 Shift-PrtSc 键时发生
6	无效操作码（80286+）
7	处理器扩展不可用（80286+）
8	IRQ0：系统时钟中断。更新 BIOS 数据区，每秒 18.2 次。用户自身的程序需要使用时钟中断时请参见 1Ch
9	IRQ1：键盘硬件中断。键盘按下的时候发生，从键盘端口读入按键并将它存储在键盘缓冲区中
0A	IRQ2：可编程中断控制器
0B	IRQ3：串行通信口（COM2）
0C	IRQ4：串行通信口（COM1）
0D	IRQ5：固定硬盘
0E	IRQ6：磁盘中断
0F	IRQ7：并行打印口
10	视频服务。控制视频显示的例程
11	设备检查。返回一个字，显示连接到系统中的所有外围设备
12	内存大小。在 AX 中返回内存的数量（以 1024 字节的块为单位）
13	软盘服务。重设软盘控制器获取最近磁盘访问的信息，读写物理扇区，格式化磁盘
14	异步端口（串行）服务。初始化、读写异步通讯信，返回端口的状态
15	磁带控制器
16	键盘服务。读取或检查键盘输入
17	打印机服务。初始化、打印以主返回打印机的状态
18	ROM BASIC。执行 ROM 中固化的 BASIC 解释程序
19	引导加载器。重启 MS-DOS
1A	每天的时间。获取自机器启动时开始的时间计数，或将该计数设为新值。计数每秒发生 18.2 次
1B	键盘中断。当 Ctrl-Break 按下的时候中断处理程序由 INT 9h 执行
1C	用户可以使用的时钟中断。空例程，每秒执行 18.2 次。可由用户的程序使用
1D	视频参数。指向包含视频控制芯片初始化信息的一张表
1E	磁盘参数。指向包含磁带控制芯片初始化信息的一张表
1F	图形字体表
20	结束程序。结束一个 COM 程序（应该尽量使用 214INT h Ch 功能代替）
21	MS-DOS 服务
22	MS-DOS 终结地址。指向父程序或进程中的地址。如果当前程序结束，则跳转到该地址
23	MS-DOS 中断地址
24	MS-DOS 关键错误地址。如果当前程序中有严重错误，比如磁盘介质错误时，则跳转到该地址
中断号	功能说明
25	绝对磁盘读（已过时）
26	绝对磁盘写（已过时）
27	结束并驻留（已过时）
28～FF	保留
33	Microsoft 鼠标。跟踪和控制鼠标的功能

续表

中断号	功 能 说 明
34-3E	浮点模拟器
3F	覆盖管理器
40～41	固定磁盘服务。固定磁盘控制器
42～5F	保留：特殊用途
60～6B	应用程序可用
6C～7F	保留：特殊用途
80～F0	保留：由 ROM BASIC 使用
F1～FF	应用程序可用

中断 21h 在 MS-DOS 服务中有许多功能，在这里仅列出一些最常用的功能。

表 F.3　常用 INT 21H 的功能调用（MS-DOS 服务）

AH	功 能 说 明
1	从标准输入读取字符。如果没有字符则等待输入。出口参数：AL=字符
2	在标准输出上显示字符。入口参数：DL=字符
3	从标准辅助输入读取字符（串口）
4	向标准辅助输出写字符（串口）
5	向打印机写字符。入口参数：DL=字符
6	直接控制台输入/输出。如果 DL=FFh，则从标准输入上读取字符。如果 DL 是其他值，则在标准输出上显示该字符
7	无回显直接字符输入。等待楞输入设备输入一个字符。出口参数：AL=字符
8	无回显字符输入。等待标志输入设备输入一个字符。出口参数：AL=字符。字符不回显，可以用 Ctrl-Break 结束等待
9	在标准输出上显示字符。入口参数：DS:DX=字符串地址
0A	缓冲键盘输入。从标准输入设备读取一个字符串。入口参数：DS:DX 指向预定义的键盘结构
0B	检查标准输入状态。检查是否有输入字符在等待。出口参数：如果字符在等待，AL=0FFh，否则 AL=0
0C	清除键盘缓冲区并调用输入功能。清除控制台输入缓冲，并执行一个输入功能。入口参数：AL=期望的功能（1，6，7，8 或 0Ah）
0E	选择默认驱动器。入口参数：DL=驱动器号（0=A，1=B）
0F～18	FCB 文件功能（已过时，可参考沈美明的书）
19	获取当前的默认驱动器。出口参数：AL=驱动器号（0=A，1=B）
1A	设置磁盘传输地址。入口参数：DS:DX 包含磁盘传输区域的地址
25	设置中断向量。将中断向量表中的表项设置为新的地址。入口参数：DS:DX 指向要插入到表中的中断处理程序地址；AL=中断向量号
AH	功能说明
26	创建新的程序段前缀。入口参数：DX=新的 PSP 的段地址
27～29	FCB 文件功能（已过时，可参考沈美明的书）
2A	获取系统日期。出口参数：AL=星期（0～6，Sunday=0），CX=年，DH=月，DL=日
2B	设置系统日期。出口参数：CX=年，DH=月，DL=日；出口参数：如果日期有效 AL=0
2C	获取系统日期。出口参数：CH=小时，CL=分钟，DH=秒，DL= 百秒数
2D	设置系统日期。出口参数：CH=小时，CL=分钟，DH=秒出口参数：如果时间有效，AL=0
2E	设置校验标志。入口参数：AL=MS-DOS 检验标志的新值（0=关闭，1=打开），DL=00h

续表

AH	功 能 说 明
2F	获取磁盘传输地址（DTA）。出口参数：ES:BX=地址
30	获取 MS-DOS 的版本号，出口参数：AL=主版本号，AH=次版本号，BH=OEM 序列号，BL:CX=24 位用户序列号
31	结束并驻留。结束当前的程序或进程，将其部分留在内存中。入口参数：AL=出口参数码，DX=请求的小节数
32	获取 MS-DOS 驱动器参数块。入口参数：DL=驱动器号。出口参数：AL=状态，DS:BX 指向驱动器参数块
33	扩展中断检查。指示 MS-DOS 是否检查 Ctrl-Break
34	获取 INDOS 标志（未公开）
35	获取中断向量。入口参数：AL=中断向量号。出口参数：ES:BX=中断处理程序的段/偏移地址
36	获取磁盘的空亲空间（只用于 FAT16）。入口参数：DL=驱动器号（0=默认，1=A 等）。出口参数：AX=每簇扇区数，如果驱动器号无效 AX 出口参数 FFFFh；BX=可用的簇数，CX=每扇区字节数，DX=每驱动器簇数
37	获取 Switch 字符（命令行参数的前导字符）（未公开）
38	获取或设置国家信息。入口参数：AL=00 或取当前国别信息=(FF 则国别代码放在 BX 中)，DS：DX=信息区首地址，DX=FFFF 设置国别代码。出口参数：BX=国别代码(国际电话前缀码)，DS：DX=返回的信息首地址。
39	创建子目录。入口参数：DS:DX 指向包含路径和目录名的 ASCII 字符串。出口参数：如果设置了进位标志 AX 中出口参数错误码
3A	删除子目录。入口参数：DS:DX 指向包含路径的 ASCII 字符串。出口参数：如果设置了进位标志 AX 中出口参数错误码
3B	改变当前目录。入口参数：DS:DX 指向含新路径的 ASCII 字符串。出口参数：如果设置了进位标志 AX，则出口参数错误码
3C	创建或剪裁文件。创建新文件或将已存文件剪裁为 0 字节。入口参数：DS:DX 指向包含文件名的 ASCII 字符串，CX=文件属性。出口参数：如果设置了进位标志 AX 中出口参数错误码；否则 AX 中出口参数新的文件句柄
3D	打开已存在的文件。打开文件进行输入/输出。入口参数：DS:DX 指向秘含文件名的 ASCII 字符串，AL=访问码（0=读，1=写，2=读/写）。出口参数：如果发生错误，设置进位标志并在 AX 中出口参数错误码；否则 AX 中出口参数新的文件句柄
3E	关闭文件句柄。关闭文件句柄指定的文件或设备。入口参数；BX=前面创建或打开的文件句柄。出口参数：如果进位标志设置，则 AX=错误码
3F	读文件或设备。从文件或设备读取指定的字节数。入口参数：BX=文件句柄，DS:DX 指向输入缓冲区，CX=要读的字节数。出口参数：如果进位标志设置，AX=错误码；否则，AX=读取的字节数
40	写文件或设备。向文件或设备指定数目的字节。入口参数：BX=文件句柄，DS:DX 指向输入缓冲区，CX=要写的字节数。出口参数：如果进位标志设置，AX=错误码；否则 AX=写入的字节数
41	删除文件。删除指定的文件。入口参数：DS:DX 指向包含文件名的 ASCII 字符串。出口参数：如果设置进位标志 AX=写入的字节数
AH	功能说明
42	移动文件指针。根据指定的方法移动文件读/写指针。入口参数：DS:DX=文件指针的移动距离，AL=方式码，BX=文件句柄。方式码如下：0 偏移相对于文件开始，1=偏移相对于当前位置，2=偏移相对于文件的末尾
43	获取/设置文件属性。入口参数：DS:DX=指向包含文件名的 ASCII Z 串，CX=属性，AL=功能号（1=设置属性，0=获取属性）。出口参数：如果进位标志被设置，则 AX=错误码
44	设备的 I/O 控制。获取或设置与打开的文件句柄关联的设备信息。或者向设备句柄发送一个控制字符串，或从设备句柄入口参数一个控制字符串
45	复制一个控制字符串，为当前找开的文件出口参数一个新的文件句柄。入口参数：BX=文件句柄。出口参数：如果进位标志设置，则 AX=错误码
46	强制复制文件句柄。强制 CX 内的文件句柄与 BX 内的文件句柄引用同一文件的同一位置。入口参数：BX=已存文件的句柄，CX=第二个文件句柄。出口参数：如果调协进位标志，则 AX=错误码
47	获取当前目录。获取当前目录的全路径名。入口参数：DS:DX 指向存放存放目录路径的 64 字节区域，DL=驱动器号。出口参数：DS:SI 指向缓冲区以路径填充，如果进位标志设置 AX=错误码

续表

AH	功 能 说 明
48	分配内存。分配请求内存，按小节数（16 字节的块）计算。入口参数：BX=请求的节数。出口参数：AX=已分配块所在的段，BX=可用的最大块（按节计算），如果设置了进位标志 AX=错误码
49	释放已分配的内存块。释放前面使用功能 48h 分配的内存。入口参数：ES=要释放块所在的段。出口参数：如果进位标志设置 AX=错误码
4A	修改内存块。修改已分配的内存块的大小。内存块可能会增大或减小。入口参数：ES=块所在的段，BX=请求的节数。出口参数：如果设置时位标志 AX=错误码，BX=最多可用块
4B	加载或执行程序。为其他程序创建程序段前缀，将其加载进内存并执行。入口参数：DS:DX 指向包含路径名的 ASCII 字符串，ES:BX 指向参数块，AL=功能值。AL 内的功能值如下：0=加载并执行程序，3=加载但不执行（覆盖程序）。出口参数：如果设置了进位标志，则 AX=错误码
4C	结束进程。结束程序出口参数到 MS-DOS 或调用程序的常用方法。入口参数：AL=8 位的出口参数码，可以用 MS-DOS 的 4Dh 功能或批处理文件中的 ERRORLEVEL 命令查询
4D	获取进程的出口参数码。获取进程或程序的出口参数码，错误码由 31h 或 4Ch 功能调用产生。出口参数：AL=程序的 8 位出口参数码，AH=产生退出的类型：0=正常退出，1=按下 Ctrl-Break 退出，2=由关键设备的错误退出，3=调用功能 31h 退出
4E	查找第一个匹配文件。查找与给定文件名匹的第一个文件。入口参数：DS:DX 指向要查找的文件名；CX=搜索时使用的文件属性。出口参数：如果设置了进位标志 AX=错误码；否则当前的 DTA 以文件的名字、属性、时间、日期、大小填充。通常在该功能调用之前要调用 DOS 功能 1Ah（设置 DTA）
4F	查找下一个匹配文件。查找与给定文件名匹配的下一个文件。应该在 DOS 功能 4Eh 调用之后调用该功能。出口参数：如果进位标志设置，AX=错误码；否则当前的 DTA 填充文件的相关信息
54	获取 verify 标志。出口参数：AH=磁盘 I/O 的 verify 标志（0=关闭，1=开）
56	重命名/移动文件。重命名文件或将其移动到其他目录中。入口参数：DS:DX 指向包含文件名的 ASCII 字符串，ES:DL 指向新的路径和文件名。出口参数：如果进位标志设置，则 AX=错误码
57	获取/设置文件的日期/时间。获取或设置文件的时间、日期戳记。入口参数：若要获取日期时间，AL=0；若要设置日期时间，AL=1；BX=文件句柄，CX=当前文件的时间，DX=当前文件的日期
58	获取或设置内存分配策略（详细信息见 Duncan 的书或 Brown 的中断列表）
59	获取扩展错误信息。出口参数 MS-DOS 错误的其他信息，包括错误类别、位置和推荐动作。入口参数：BX=MS-DOS 版本号（版本 3.xx 为 0）。出口参数：AX=扩展错误码，BH=错误类别，BL=建议动作，CH=位置
AH	功能说明
5A	创建临时文件。在指定目录内生成一个独一无二的文件。入口参数：DS:DX 指向以反斜线（\）结尾包含路径的 ASCII 串；CX=期望的文件属性。出口参数：如果设置了进们标志，AX=错误码；否则，DS:DX 指向创建文件的全路径
5B	创建新文件。试图创建新文件，但是如果文件名已经存在则失败，这防止了覆盖已存在的文件。入口参数：DS:DX 指向包含文件名 ASCII 串。出口参数：如果设置了进位标圮，则 AX=错误码，否则 AX=文件句柄
60	扩展为全路径文件名。入口参数：DS：SI=ASCII_ZERO 串的地址，ES：DI=工作缓冲区地址。出口参数：如果失败 AX=错误代码。
62	获取程序段前缀（PSP）地址。出口参数：BX=当前程序段前缀的段值

参 考 文 献

1．Barry B.Brey.Programming the 80286,80386,80486,and Pentium-Based Personal Computer. Prentice Hall,Inc.1996
2．Peter Abel.IBM PC Assembler Language and Programming.Prentice Hall,Inc.1987
3．王保恒. IBM PC 宏汇编语言程序设计及应用. 长沙：国防科技大学出版社.1993
4．沈美明等. IBM-PC 汇编语言程序设计. 北京：清华大学出版社.1999
5．王元珍等. IBM-PC 宏汇编语言程序设计. 武汉：华中理工大学出版社.1990
6．周学毛等. 80486（80x86）汇编语言程序设计. 北京：电子工业出版社.1997
7．杨季文等. 80x86 汇编语言程序设计教程. 北京：清华大学出版社.1998
8．潘名莲等. 微机原理与应用. 成都：电子科技大学出版社.1995
9．罗云彬. Windows 环境下 32 位汇编语言程序设计. 北京：电子工业出版社
10．Kip R. Irvine（温玉杰等译）. Intel 汇编语言程序设计（第四版）. 北京：电子工业出版社
11．殷肖川等. 汇编语言程序设计. 北京：清华大学出版社.2005
12．钱晓捷. 新版汇编语言程序设计. 北京：电子工业出版社